High-Performance FPGA Network Switch Architecture

Philippos Papaphilippou, Jiuxi Meng, Wayne Luk
Department of Computing, Imperial College London, UK
{pp616,jiuxi.meng16,w.luk}@imperial.ac.uk

ABSTRACT

We present a high-throughput FPGA design for supporting high-performance network switching. FPGAs have recently been attracting attention for datacenter computing due to their increasing transceiver count and capabilities, which also benefit the implementation and refinement of network switches. Our solution replaces the crossbar in favour of a novel, more pipeline-friendly approach, the "Combined parallel round-robin arbiter". It also removes the overhead of incorporating an often-iterative scheduling or matching algorithm, which sometimes tries to fit too many steps in a single or a few FPGA cycles. The result is a network switch implementation on FPGAs operating at a high frequency and with a low port-to-port latency. It also provides a wiser buffer memory utilisation than traditional Virtual Output Queue (VOQ)-based switches and is able to keep 100% throughput for a wider range of traffic patterns using a fraction of the buffer memory and shorter packets.

KEYWORDS

Network switch, FPGA, round-robin, arbiter, scheduling algorithms, sorting network applications, stream processing

ACM Reference Format:
Philippos Papaphilippou, Jiuxi Meng, Wayne Luk. 2020. High-Performance FPGA Network Switch Architecture. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3373087.3375299

1 INTRODUCTION

While FPGAs have already been adopted by cloud providers as a hardware acceleration platform, FPGA vendors have been increasing the transceiver count on mid to higher-end FPGAs, making them attractive also for single-chip network switch implementations. Xilinx Virtex UltraScale+ FPGAs support up to 48 GTM transceivers (up to 58 Gbps each), and up to 128 GTY transceivers (up to 32.75 Gbps each), depending on the speed grade and device/package combination [14, 15]. In the latest device iterations, the available logic resources have also increased substantially. For example, the recently announced VU19P device features around 9 million logic cells. This leaves a gap in research on FPGA-based switches, which are usually unable to saturate the bandwidth of the transceivers while retaining competitive switching performance.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375299

FPGA-based network switch implementations are inspired by well-known approaches used for ASICs/ASSPs, usually classified by the position of their buffer memory. Output-queued (OQ) network switches have been shown to perform better than input-queued (IQ) switches [8]. However, their adoption has been limited, due to the requirement for output queues to operate at a much higher frequency. To solve this problem, various hybrid approaches emerged, making use of both input queues and other memories [9]. These present different shortcomings, such as the need to move the speedup to the internal logic. Moreover, for FPGAs such workarounds have a prominent performance overhead, being unable to saturate the available link bandwidth efficiently. Thus, the latest FPGA solution has fallen back to input-queued switches [22], and with bigger flits, in order to achieve a moderately high throughput, inheriting the related limitations.

This paper presents a novel network switch designed to perform well as an FPGA-based implementation, since it does not require iterative algorithms [22], logic speedup [9] or memory speedup. At the same time we show that it is competitive as a switching approach under different traffic patterns. This is something that previous research on FPGA switches gave almost no emphasis, despite its impact on memory utilisation, dropped packet rate and average latency.

As a proof of concept, our *open-source* implementation of a 16×16 switch provides an aggregate throughput of 903.2 Gbps on a Zynq UltraScale+ FPGA, surpassing the last reported, theoretically achievable 819 Gbps [1], which requires an embedded network-on-chip (NoC) [5].

Our key contributions in this work are as follows:

- A high-throughput network switch design on FPGAs, based on a sorting network.
- A new buffer topology with better performance, that requires a small fraction of the memory used in today's Virtual Output Queues (VOQs).
- A modular design approach that focuses on high throughput and a technique to significantly reduce the port-to-port latency.
- Extensive simulation results comparing the algorithmic aspect of our approach to a selection of scheduling algorithms.

2 BACKGROUND

2.1 Head-of-line (HOL) blocking

HOL blocking is the performance bottleneck in crossbars when multiple inputs request to send to the same output port. It is known to limit the throughput to just 58.6% [21]. The available workarounds try to appropriately rearrange the transmissions of the most recently arrived packages using temporary buffer memories.

2.2 Virtual Output Queues (VOQs)

VOQs are widely accepted as a solution to head-of-line blocking. They are used in input-queued (IQ) switches as a buffer space between the inputs and the crossbar. There is one queue for each source/destination combination, totalling $P \times P$ queues, where P is the number of ports. The idea is that a packet will always have a place to go, until its transmission is determined by a *scheduling algorithm*, given that the respective queue is non-empty. As demonstrated in figure 1 (and in section 5), VOQs have performance limitations. These include buffer fragmentation, causing dropped packets, and poor performance for bursty and nonuniform traffic.

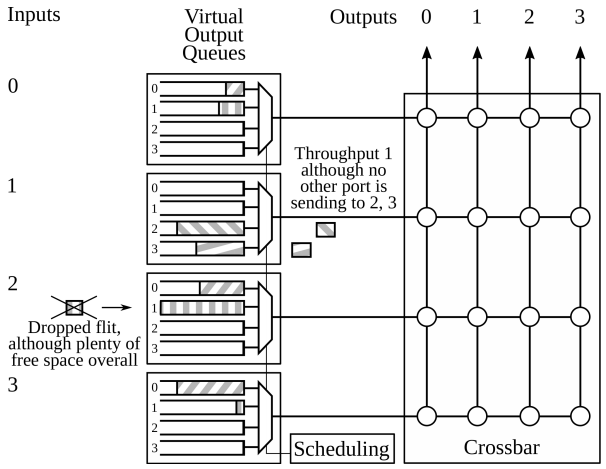

Figure 1: An input-queued switch showing two limitations

They are still found today in the highest-end switches, such as Arista's 7800R3, which provides an aggregate throughput of up to 460 Tbps [3]. VOQs also exist in some form in hybrid memory model switches [9], such as in combined input and output queued (CIOQ [8]), combined input and crosspoint queued (CICQ [16]) and hierarchical crossbar (HC [17]) switches.

2.3 Scheduling algorithms

Scheduling or matching algorithms are used to make those 'dequeue' decisions, by finding a nearly-maximum matching between the available input and output ports [10, 21]. The goal is to maximise throughput, while providing other desirable features, such as preventing starvation, i.e. ensuring no input is getting blocked indefinitely. They require multiple iterations to perform well for various traffic patterns.

2.4 Round-robin arbiter

A round-robin arbiter is used for input arbitration to achieve fairness to incoming requests [21]. It is commonly applied to dequeue packets from 1 out of P queues, with a round-robin priority. The incoming packets can arrive at any of the P inputs. When there is an input available, a grant signal is enabled, and the arbiter selects the next available queue, based on a priority offset.

In hardware, this is usually implemented using a programmable priority encoder (PPE). The PPE can be expensive to implement efficiently and can contribute to the critical path for an increasing number of inputs [11].

In practice, the round-robin arbiter can have small modifications to accommodate the requirements of a scheduling algorithm. One way is to postpone the rotation until a queue is emptied entirely, in order to improve scheduling performance for bursty traffic [19].

2.5 Parallel round-robin arbiter

The parallel round-robin arbiter [27] receives from 0 to P packets per cycle, and rearranges them so that the output in P memory banks appears as it is written in round-robin fashion. In contrast to a round-robin arbiter, which has a maximum output rate of 1, it is a pipeline-friendly stream processor with a throughput of P packets per cycle. As every packet is processed at line-rate, there is no need for additional memory for producing the round robin effect. Figure 2 shows an example of input and output of such an arbiter, capable of processing multiple requests at a time. The input involves packets arriving in arbitrary positions.

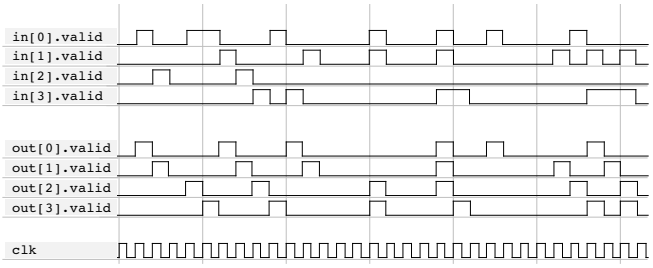

Figure 2: Parallel round-robin arbiter example, P=4

When this output is stored in banked memory, it has the advantage of using memory efficiently, as the data are evenly distributed across the P banks. Afterwards, the stored output data can be serialised by dequeuing in round-robin fashion, ensuring fairness and preventing starvation.

3 PROPOSED DESIGN

The main idea is to use fully-pipelined logic to process the incoming packets at line-rate and insert them in output queues in a balanced way. There are P groups of P output queues, where P is the number of ports. Similar to Virtual Output Queues (VOQs), this requires $P \times P$ queues. However, by moving those queues towards the output, the traffic has the opportunity to get reorganised in such a way, that each of the P FIFO groups is filled with packets only destined for that port. In each cycle, up to P packets arrive, which are then grouped according to their destination. These packets are permuted such that the P queues in each of the P groups appear to be filled serially in round-robin fashion, but this is done in parallel, as with a parallel round-robin arbiter. Figure 3 outlines this architecture.

The balanced traffic approach achieves better utilisation of the available memory resources. This is because the P queues of the same group are filled at the same rate, leaving no gaps of unused resources. In contrast, the $P \times P$ queues in traditional VOQs assign only one queue for each $(port_{source}, port_{dest.})$ combination. A packet arriving in our proposed switch has the possibility to land in P queues for $port_{dest.}$, which are virtually unified.

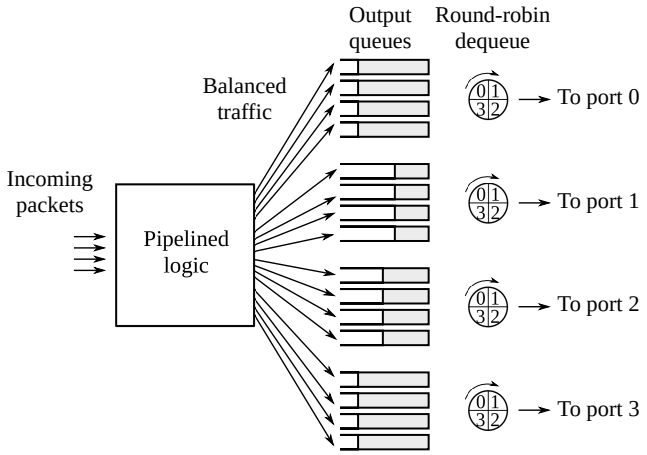

Figure 3: Abstracted view of the proposed solution, P=4

Apart from its advantageous behaviour as an algorithm, our design leads to efficient hardware implementation. The proposal replaces the crossbar in favour of more pipeline-friendly structures. This removes the need to solve bipartite graph problems (or approximations) on-the-fly, which is what scheduling algorithms are for.

Near the ports, we have P *round-robin arbiters*, which rotate to provide the content of each FIFO group in a serialised manner, with a throughput of 1 per port.

3.1 Combined parallel round-robin arbiter

The main element in our solution is the *Combined parallel round-robin arbiter* (Figure 4). It can be seen as a hardware-optimisation of the following switching scheme. The functionality is equivalent to having P parallel round-robin arbiters, one for each output port, for balancing its FIFOs occupancy. This holds because the output of the pipeline for each port essentially does not depend on the packets destined for the other ports. As a result, the proposal only uses one sorting network, instead of P sorting networks for P separate *parallel round-robin arbiters*, hence its name.

It consists of two pipelines being merged together, with additional pipeline stages at the end for the final rotation of the packets, before appending them to the FIFO group of their destination port. The first pipeline is for the 'sorting network' and the second one, 'offsets', is for calculating the amount of rotation each barrel shifter (rotator) needs to perform. The 'offsets' pipeline includes everything below the sorting network in figure 4. The two pipelines have different lengths and therefore additional shift registers are used for synchronising their output.

The arbiter architecture is fully pipelined and processes up to P packets at line rate. At its output, all inserted packets are distributed into the output queues. The pipeline logic is non-blocking, and any dropped packets can only occur at the queue level.

3.2 Building blocks

Here, we provide an overview of the building blocks, and how they interact with each other. Figure 5 shows examples of the main building blocks (**sorting network**, **adder tree**, **prefix sum** and **barrel shifter**) for a specific input size, combined with registers, to work as pipelines in hardware.

a) Sorting network (Batcher's odd-even mergesort). This structure consists of compare-and-swap (CAS) units and is able to sort a list of P elements in a number of parallel steps. The CAS units can be seen as sorters of 2 elements. There are two topologies frequently used in FPGAs [23]: Batcher's odd-even merge sort and bitonic sort [4]. They have the same pipeline length, which is $(log_2(P) \cdot (log_2(P) + 1))/2 \in \Theta(log^2(P))$ steps. However, we found out that Batcher's odd-even merge sort maps better on FPGA resources, and this is because it has shorter total wire length and fewer CAS units than bitonic sort. Such structures are straightforward to produce for any value of P, though more optimal networks exist [6]. Depth-optimal sorters are difficult to find for large P-values.

In our proposal, the sorting network is used to sort a batch of P packets according to their $valid$ and $dest.$ fields. The goal is to have all packets going to the same port arranged in consecutive order, before feeding them to the rotators. The sort priority is given to the invalid packets, so the comparison in the CAS units is done on the concatenation $\{valid, dest.\}$, which is $1 + log_2(P)$ bits wide.

b) Adder tree. This structure calculates the sum of P inputs (i.e. $out = \sum_{i=0}^{P-1} in_i$) in $log_2(P)$ parallel steps. In the pipelined version, as shown in figure 5, each step has a number of independent adders that work in parallel and their results are saved in pipeline registers.

In our design, the adder trees are used to perform popcount, i.e. to count the number of packets coming in a batch of up to P, that satisfy certain criterion. We have P+1 adder trees for P inputs. The first tree counts the number of packets with their valid bit unset ($Null$ or non-existent). Each one of the other P trees are counting the number of packets going to each of the P ports. This information is later used by the prefix sum, as well as to update the *offset counters* (keeping the 'current' write rotation index of the output queues).

c) Prefix sum. The parallel version of prefix sum [13] can be seen as a superset of the adder tree. It provides the cumulative sum (i.e. $out_k = \sum_{i=0}^{k} in_i$ for $k \in \{0, 1, ..., P-1\}$) in $log_2(P)$ parallel steps.

The prefix sum is used here to identify the starting positions of each group of packets going to the same destination port, within the sorted batch of P packets. The inputs of the prefix sum are the P out of the P+1 popcounts, starting from the $Null$ count for in_0, the $(dest. == 0)$ count for in_1, and up to the $(dest. == P-2)$ count for in_{P-1}. Since we give priority to $Null$ packets in sorting, the out_0 wire of the prefix sum represents the start index of the packets going to $port_0$, or the index just after the $Null$ packets. Those starting positions are used to calculate the final amount of rotation each copy of the sorted batch of P packets needs, before writing them to the FIFO groups, in order to achieve the round-robin effect.

d) Barrel shifters (rotators). The purpose of the rotator is to rotate a number of elements by a specified offset. The wiring in our pipelined implementation, as shown in figure 5d, has a small resemblance to the wiring of prefix sum and consumes $log_2(P)$ pipeline stages. The select signal gets propagated alongside the data (packets) and the respective bit is fed across multiple 2-to-1 multiplexers in each stage.

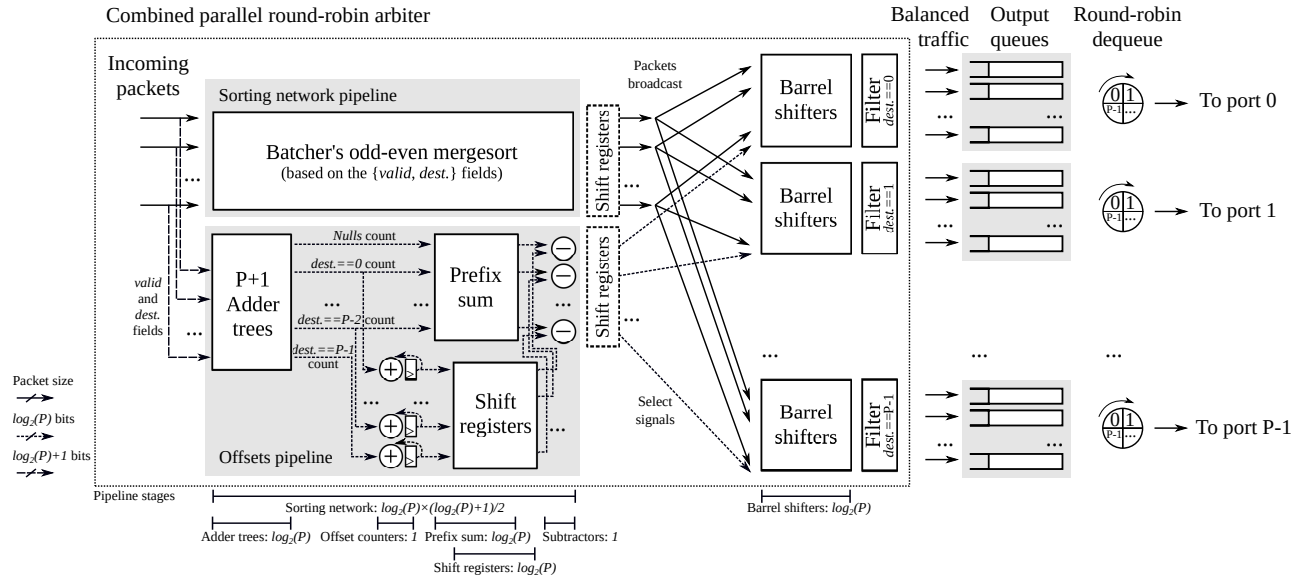

Figure 4: The proposed high-performance FPGA network switch

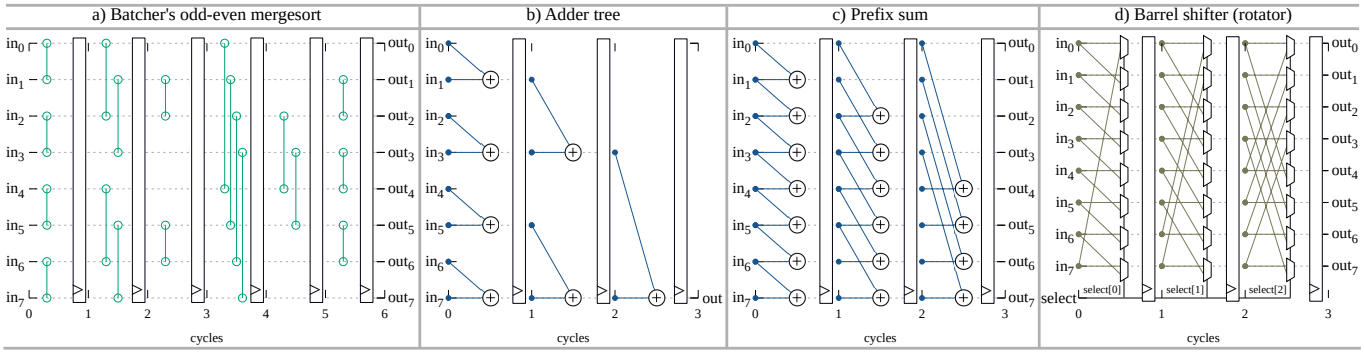

Figure 5: Pipelined versions of the building blocks for 8 inputs

The rotators are useful for performing the final rotation on the batch of 0 to P consecutive packets destined to each port. Note that the rotations are done on all packets in the batch, as it is broadcasted to all rotators. The output of each of the rotators is filtered afterwards to ensure that it only serves the respective destination port.

Filters. In our design, the filters check the validity of a packet or whether the destination field equals to a specified port number (an integer between 0 and P-1). Therefore, an equality check is performed for only up to $log_2(P)$ bits, which has little timing slack. Therefore, we have not assigned any pipeline stages for them.

Subtractors. The subtractors occupy one pipeline stage and their purpose is to produce the select signals for the rotators. Each rotator performs a rotation equivalent to two rotations, one to the left, for bringing the packets going to the respective port to position 0 in the sorted packet batch, and one to the right, to achieve the round-robin effect. The subtractor calculates the addition of the positive offset to the negative offset. The positive offset corresponds to the (delayed) queue rotation offset for the port, produced by the *offset counters*. The negative offset corresponds to the respective output value of the prefix sum, i.e. out_0 the start index for packets to $port_0$, etc.

4 PROPERTIES AND OPTIMISATIONS

4.1 Complexities

In terms of hardware resource utilisation, the fastest growing of all building blocks is the sorting network. This highlights the importance of the "combined" aspect of the *Combined parallel round-robin arbiter*, whose reduced hardware resource utilisation makes its implementation practical on FPGAs. The hardware complexity of each odd-even merge sorting network is $O(P \times log^2(P))$ CAS units. The reduction to just one sorting network reduces the overall hardware complexity to that of the P barrel shifters, which sums to $O(P^2 \times log(P))$ 2-to-1 multiplexers.

The pipeline length (*latency*) is one of the main contributors to the port-to-port latency, as it is added as a processing latency,

even when the output queues are empty. It is calculated as the sum of the barrel shifter latency ($latency_{BS}$) and the maximum of the two pipelines at the beginning, which are the sorting network ($latency_{SN}$) and the offset calculation logic ($latency_{offsets}$).

$$latency = latency_{BS} + max\ (latency_{SN},\ latency_{offsets})$$
$$= log_2(P) + max\ (\frac{log_2(P)\times(log_2(P)+1)}{2},\ 2\times log_2(P)+1)$$
$$\in O(log^2(P))$$

As it is, the resulting port-to-port latency is not competitive. However, we now introduce an optimisation to improve the latency significantly, sometimes with a small overhead in operating frequency.

4.2 Reducing the port-to-port latency

The logic complexity of each stage, although highly parallel, it is lightweight. This is because the basic components are comparing, adding or subtracting up to $log_2(P)$+1 bits. This allows to combine multiple pipeline stages in one cycle.

The pipeline latency can be reduced by removing the registers from all pipeline stages whose order is not multiples of S, where $S \in \mathbb{N}^*$ can approximately divide the port-to-port latency (i.e. $latency_{opt} \approx latency/S$). In this way, although the complexity of the number of pipeline stages is more than that of competing solutions, our approach can still provide lower port-to-port latency for moderately high values of P. An example for dividing a sorter's latency by 3 is shown in figure 6.

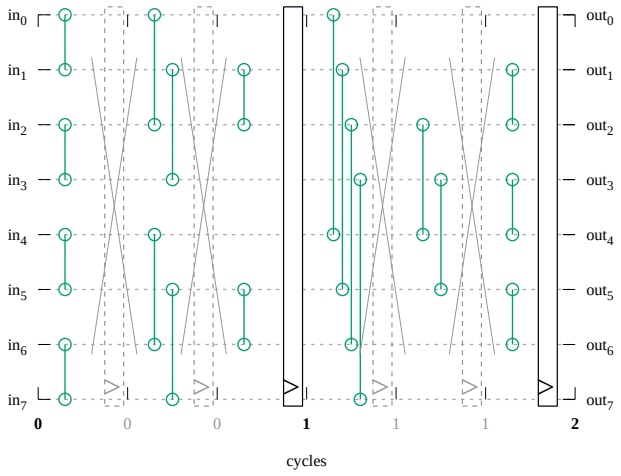

Figure 6: Example for dividing the latency of a sorter by S=3

At the programming level, when there is a shift register corresponding to a stage without registers and $S > 1$, we replace the shift register with a wire. This is done for the sake of simplicity, by always keeping the notion of stages. For example, a shift register of length 4 and S=2 is essentially implemented as a shift register of length 2.

The only pipeline stage whose registers cannot be removed is where the P adders are keeping the queue offsets for the P ports, as shown in figure 4. This is because these adders are the only place where a feedback exists in our pipeline. The output of each adder is added back as a way to update the respective offset. Fortunately, this feedback is always one stage long and does not contribute to the critical path, but adds development complexity to this optimisation. We denote the *compulsory register index* as $index_{comp}$ and it represents the index of this pipeline stage (starting from 0).

$$index_{comp} = log_2(P).$$

Keeping the above in mind, the optimised pipeline latency ($latency_{opt}$) is calculated as follows.

$$latency_{opt} = \lceil \frac{latency - index_{comp}}{S} \rceil + \lceil \frac{1+index_{comp}}{S} \rceil - 1$$

Table 1 illustrates how this optimisation significantly reduces the port-to-port latency. Illustratively, one more cycle is added to the port-to-port latency, representing the minimum time a packet stays in the output queues. A typical operating frequency of 150MHz was selected to enable translating the pipeline latency from cycles to nanoseconds, before the actual place-and-route in the evaluation (section 5.2).

Number of ports (P)	2	4	8	16	32	64	128	256
Pipeline length	4	7	10	14	20	27	35	44
Compulsory reg. index	1	2	3	4	5	6	7	8
Pipeline latency, S=1	4	7	10	14	20	27	35	44
Pipeline latency, S=2	2	4	5	7	10	14	17	22
Pipeline latency, S=4	1	2	2	4	5	7	8	11
Port-to-port latency (ns) (*indicative*, with S=4 and f_{clk}=150MHz)	13.3	20	20	33.3	40	53.3	60	80

Table 1: Indicative port-to-port latency after reg. reduction

An unexplored optimisation is to use a different values of S for different pipeline stages groups. An example for stages with more complexity can be seen in figure 5, where the most dense of our building blocks in terms of wiring are the barrel shifters.

In other applications, different pipelining techniques may be more appropriate. If we restrict this discussion on sorting networks on FPGAs (with no instructions), there are three common pipelining techniques: (a) *synchronously*, using up to $P/2$ comparators per stage, (b) *asynchronously*, incorporating additional logic for evaluating completion [23] and (c) with *unary processing* [24], increasing the latency. Our use-case-specific solution lies between (a) and (b), as it is still a synchronous circuit but can achieve a performance closer to the asynchronous, without the related complications.

Section 5 contains more details about the impact of this optimisation on Flip-Flop register utilisation, as well as on operating frequency. Additionally, using simulation, we measure the average time a packet stays in the queues using different approaches and traffic patterns, as the latency here represents only the minimum.

4.3 Pipeline synchronisation

Our description includes an additional group of shift registers, for either the 'sorting network' pipeline or the 'offsets' pipeline, depending on which one is the shorter. The number of stages in the shift registers is equal to the pipeline length difference between the the two pipelines (i.e. $|(log_2(P)\times(log_2(P)+1)/2) - (2\times log_2(P)+1)|$).

An optimisation to reduce the register utilisation can be achieved where the 'offsets' pipeline is shorter than the sorting network (for $P > 8$). In such cases, the 'offsets' pipeline can start at a later stage, by reading the $valid$ and $dest$. fields from a later stage of the sorting network. This is possible due to the fact that during the sorting network pipeline stages, a set of up to P packets is only getting permuted, and the fact that the addition operation is commutative and associative. This optimisation removes the need for additional registers as a synchronisation measure when $P > 8$, and is included in all our implementations (but not reflected in table 1).

Another possible optimisation for $S > 1$ and $P > 8$ is to position the compulsory register stage in a position that could benefit the total number of stages with registers (and therefore the total pipeline latency). The compulsory register stage can be moved by selecting a different start for the 'offsets' pipeline, and introducing additional shift registers where required. This can only save up to 1 cycle in latency. At a high level, the idea is to keep the maximum allowable consecutive stages without registers, at the end and the start of the pipeline, so that the register removal optimisation becomes more beneficial.

4.4 Simpler output arbiters

The output arbiters can be simpler than traditional *round robin arbiters*. This is because only one queue is needed to be checked for available packets, as the queues in a FIFO group are filled in round-robin fashion as well. At decision time, the queue to be checked is the one denoted by the arbiter's index (rotating in the range from 0 to $P - 1$). Originally, round robin arbiters produce a *grant* signal, which relates to P different input queues, potentially becoming the critical path for large P, assuming everything else is efficiently pipelined.

This optimisation assumes no packets are being dropped, otherwise there is a small performance overhead, as the output arbiter can get out of synchronisation. Our solution includes this optimisation, as the overhead of doing so is negligible for all traffic models in our simulations.

4.5 Output per cycle

The *Combined parallel round-robin arbiter* produces 0 to P packets in every FPGA cycle, without depending on algorithm phases or iterations. Also, as with modern designs for high-throughput data processing [25, 26], it can be considered 'feedback-less', as no input of a pipeline stage depends on the output of a subsequent stage. Given that the line rate supports it, this means that the operating frequency and packet size combination can be directly translated into the maximum throughput.

This is not the case with iterative algorithm approaches, such as iSLIP-based implementations [11, 22]. Essentially, our solution can support many times more throughput for the same packet size than iterative approaches.

5 EVALUATION

Our proposal is first implemented in software for high-level simulation, in order to study its performance as a switching algorithm. Once successful, an RTL generator script is implemented to produce Verilog code for a switch of any specified packet size, number of ports (P) and latency reduction (S).

The generated RTL code is validated in two ways: (a) using *RTL simulations* with a testbench and waveforms for observing the parallel round robin effect, and (b) in *real hardware* on an Avnet's Ultra96 featuring the ZU3EG device and running Linux.

In the latter case, the generated Verilog code is encapsulated in an AXI peripheral with the following debugging functionality: a batch of up to P packets is stored serially into registers by the user's software. A 'release' command is sent to insert all packets in the pipeline, during the same FPGA cycle. A similar procedure is followed for retrieving the packets back in software to verify that the outcome is the same as that of RTL or non-RTL simulations.

All source code used for evaluation, including the RTL code and the simulators for competitor algorithms and traffic patterns, has been developed from scratch and are freely available[1].

5.1 Simulation: Algorithm evaluation

This part of the evaluation is conducted in software (non-RTL) and ignores hardware-related complications, such as the pipeline latency, bandwidth and packet size. The idea is to focus on the algorithmic aspect of our proposal. Therefore, we assume that the algorithms themselves run on a switch at the same operating frequency/ bandwidth and introduce an unrealistic processing latency of 0 cycles.

We consider our approach an alternative to scheduling algorithms for crossbar switches, even though we have replaced the crossbar altogether. This is because our proposal uses $P \times P$ queues, as in VOQs for input-queued switches. Another similarity is that our solution requires no speedup of any memory or logic, as a single clock is used for the entire design. Due to their simplicity, one of them is also incorporated in the state-of-the-art FPGA network switch implementation [22], outperforming a solution with a hybrid memory model [9].

The performance of the proposed design is compared against various scheduling algorithms. We present comparison results for parallel iterative matching (PIM [2]), round-robin matching (RRM [21]), iSLIP [21], dual round-robin matching (DRRM [7, 18]) and dual round-robin matching switch with exhaustive service (EDRRM [19]).

The progress in such algorithms tends to be incremental, in the sense that each approach improves on some aspects of a predecessor. The changes focused on improving ASIC implementation efficiency [18, 21], scheduling performance and/or their behaviour under a wider spectrum of traffic patterns [12, 19].

Although all can work as iterative algorithms, some of them are originally presented as 1-iteration approaches, due to their simplicity and relative-efficiency. Regardless, we consider their single-iteration version, as well as with multiple ($log_2(P)$) iterations.

With respect to the traffic, the proposed approach is evaluated for the following patterns:

(a) *Uniform Bernoulli arrivals:* At each input port, in each cycle a packet is sent with a probability r (input rate). The destination port of each packet is any of the P ports, all appearing with an equal probability $1/P$.

[1] **Source code available:** http://philippos.info/switch

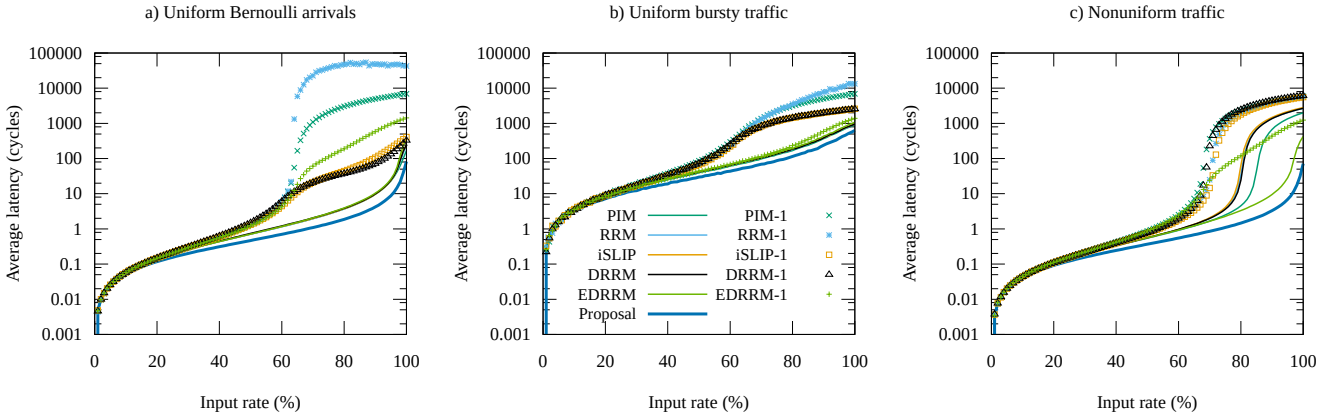

Figure 7: Simulated average packet latency using different algorithms and traffic models (16×16 switch)

(b) *Uniform bursty traffic:* The packets at each input are sent based on an independent Markov chain, which has three states: 'On', 'Off' and 'New'. In each cycle at 'Off', no packet is emitted. At 'New', a packet is sent with its output destination selected uniformly among the ports. At 'On', a packet is sent to the same destination as the one of the previous packet. The transition probabilities are shown in figure 8, where r is the average input rate and s is the average burst size. The proposed model is almost equivalent to the two-state model [12], but it does not limit the maximum input rate to $1 - \frac{1}{s+1}$. The default value of s is set to 32.

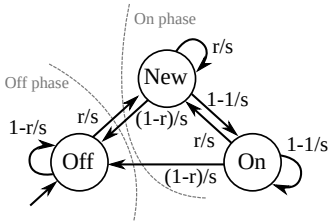

Figure 8: A uniform bursty traffic Markov model

(c) *Nonuniform traffic [12, 19]:* At each input, in each cycle a packet is sent with a probability r. Its destination will be the same as the source with a probability p. All other outputs share an equal probability, i.e. $(1 - p)/(P - 1)$. The default value of p is set to 0.5.

Average latency. The first experiment explores the average latency a packet waits inside a queue to be served. In this experiment we assume that the $P \times P$ FIFOs have *infinite depth* and, thus, no dropped packets.

As we can see in figure 7, the proposed design consistently outperforms the scheduling algorithms studied here. For each of the scheduling algorithms, we include a multi-iteration version and a single-iteration version, denoted by appending "-1" to its name. For both the traffic patterns (a) and (b), the multi-iteration scheduling algorithms perform almost identically to each other. (It is their single-iteration performance that motivated most of them). In traffic model (c), the only close competitor to our approach is EDRRM [19], which is selected here as an algorithm designed for bursty/nonuniform traffic. Still, for an input rate of 1, our proposed approach reduces the average latency by 2x, 1.4x and 6x for the traffic models (a), (b) and (c) over the best alternative for each (PIM, PIM and EDRRM).

Worst-case latency. Using a similar experiment, for a 16x16 switch, we observe that our approach also yields the lowest average worst-case latencies. The best alternative algorithms remain the same for this metric. Using an input rate of 100%, the best alternative per traffic model yields a latency of 1535, 7992 and 7212 cycles for the traffic patterns (a), (b) and (c). The corresponding values for our proposal are 375, 2807 and 310 cycles, which translate to a reduction of the worst-case latency by 4.1, 2.8 and 23.3 times on average respectively.

Impact of queue size. As a third experiment, the queue size requirements are explored. The best alternative algorithms are still PIM, PIM and EDRRM for the traffic patterns (a), (b) and (c) respectively. The approach is to measure the dropped packet rate using different FIFOs lengths. The FIFO depth metric is a consistent way to measure the total buffer size in our simulations, as both virtual output queues (VOQs) and our proposal's output queues are $P \times P$ buffers.

Figure 9 shows the dropped packet rate with respect to the queue size for our approach, as well as for the best alternative for each of the traffic patterns. Our approach requires a small fraction of memory to support 100% throughput, i.e. a dropped packet rate equal to zero (represented by the colour white). For the traffic pattern (a) and an input rate 1, we achieve a 4.6x reduction in FIFO depth (153 to 33) for supporting 100% throughput, while the dropped packet rate for a FIFO depth of 1, drops from 16.1% to 5.2%. For the traffic patterns (b) and (c) and input rate 1, the other approaches require a FIFO depth beyond our design space, which is a sign for not supporting the 100% throughput, as the memory's order of magnitude approaches the one of the number of inserted packets. Nevertheless, our approach exhibits a better behaviour.

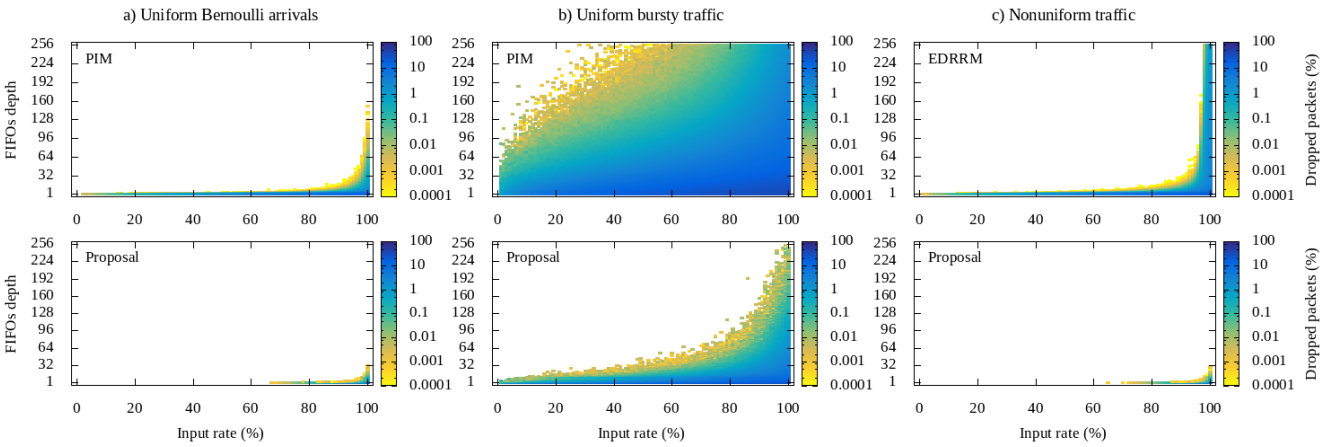

Figure 9: Impact of queue size, for the proposal and the best alternative per traffic model (16 × 16 switch)

Moreover, for the traffic pattern (c), it is able to handle nonuniform traffic with marginally less requirements than in the Bernoulli case, while this pattern becomes much more challenging for the other approaches.

5.2 FPGA implementation

This part of the evaluation explores the performance and scalability of our design on FPGAs, using the RTL code produced by our Verilog generator.

First, the resource utilisation in flip-flop registers (FF) and lookup tables (LUT) is presented, as reported by Vivado 2018.3. Figure 10 shows these numbers for a switch design space with 256-bit packets, the number of ports ranging from 2 to 32 (powers of 2) and the latency reduction parameter S for the values of 1, 2 and 4.

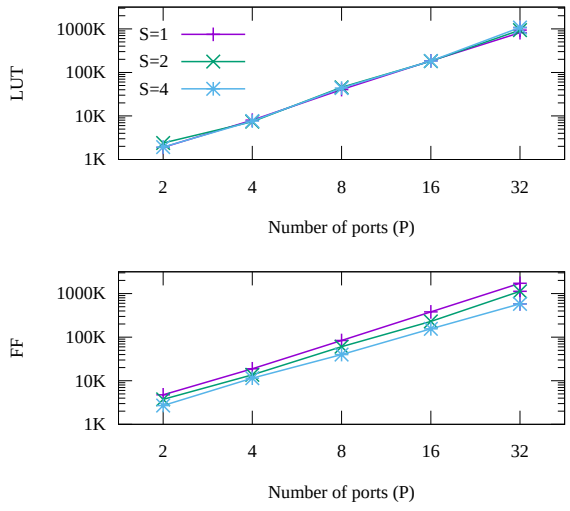

Figure 10: Synthesis results for a packet size of 256 bits

It can be seen that the flip-flop utilisation is significantly decreased by the register reduction optimisation. For example, for P=32 and 256-bit packets, if we use approximately one quarter of the pipeline registers (using S=4), the FF utilisation drops from 1720K to 579K. This is not the case for the lookup tables, as the functional units remain more or less the same, with some variations due to tool-related optimisations.

In order to restrict our design space and focus on the more demanding switch logic, the FIFO queue depth is limited to one, implemented as registers. The effects of different memory technologies and hierarchies for different devices and number of ports are beyond the scope of this paper. It is worth noting that a shallow queue depths could be a realistic use case, as our approach performs similarly to using VOQs with deeper FIFOs, as shown in our simulation results. In many-port switches, register/LUTRAM-based queues become a viable solution to the block RAM limitation explored in related research [22] (VOQs as BRAM require many multiples of $P \times P$ blocks for the common packet widths). The reduction of memory needs in our solution can provide finer resource granularity and thus, feasibility.

As a target device we select the Zynq UltraScale+ ZU19EG, having the same architecture as ZU3EG, that was used for validation. The resources on ZU19EG are comparable to those in mid-range UltraScale+ FPGAs, which commonly appear in today's research on accelerator design. As a use case example, this device is also available in the package FFVC1760, which provides 16 GTY transceivers, with 32.75 Gbps per link. A goal would be to allow saturating these links using the available programmable logic for implementing a 16 × 16 switch.

The performance results for this device are summarised in figure 11. There are 256-bit and 512-bit-wide versions of 2, 4, 8 and 16-port switches. Each of those 8 data series has 3 data points, one for each S value in {1, 2, 4}. The two performance metrics here are the line throughput and port-to-port-latency, which are measured in the y and x axis respectively. The throughput is calculated as the product of the operating frequency and the packet size (i.e. $Throughput = Width \times f_{clk}$). The port-to-port latency is the pipeline latency in cycles, plus one extra cycle for reading from the queues, multiplied by the clock period (i.e. $latency_{p2p} = (latency_{opt} + 1)/f_{clk}$).

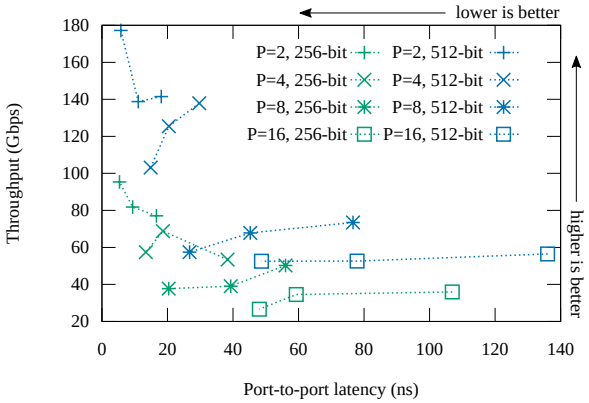

Figure 11: Throughput and port-to-port latency on ZU19EG

For S=1 and 256-bit packets, the maximal operating frequency equals to 301, 208, 196 and 140 MHz for the designs with 2, 4, 8 and 16 ports respectively. For 512-bit packets, the corresponding values are 276, 269, 143 and 110 MHz. When moving to S=4, the operating frequency changes slightly, such as from 110 to 102MHz for the 16x16 512-bit switch.

As shown in figure 11, our design almost always exceeds the throughput of GTY (32.75 Gbps). As expected, at least for $P > 4$, the latency optimisation has a small overhead on throughput, but significantly reduces the port-to-port latency. In the following section we select our 16×16 switch implementations for comparison to the related work.

6 RELATED WORK

An FPGA network switch implementation is GCQ [9]. It improves on a hybrid buffer approach, the hierarchical crossbar (HC) switch [17]. The idea is to improve the scalability for a higher number of ports by using smaller switches hierarchically. Due to the fact that it requires a logic speedup, the yielded base frequency was 40 MHz, resulting in an aggregate throughput of 160 Gbps for a 16×16 switch. For a better comparison with today's standards, the same design was reimplemented on an UltraScale+ device (VU9P) [22], and its aggregate throughput value is upgraded to 222 Gbps, using a data width of 256 bits.

The most recent switch implementation on FPGA is SMiSLIP [22]. It is an IQ switch, implemented as a crossbar with VOQs, and controlled by the iSLIP scheduling algorithm [21]. The focus of this work is the optimisation for FPGAs to achieve buffer-sharing and a better performance. Despite the relatively high operating frequencies, SMiSLIP only produces output once every $3 + log_2(P)$ cycles. For this reason, the throughput is calculated as $\frac{1}{3+log_2(P)}$ of the line speed. The aggregate throughput for a 16×16 switch is 118, 282 and 538 Gbps, for 256, 640 and 1280-bit packets respectively.

This is expected for iterative matching algorithms, such as iSLIP. One of the original iSLIP-based prototypes [11] is an ASIC operating at 175MHz in a 32-port switch. Despite its optimisations such as the merging of the 2 out of its 3 phases (request, grant and accept), it is not fully-pipelined. It only produces output once every 9 cycles, reducing the maximum throughput to only 1/9 of the speed the fabric. It does not necessarily mean that the lines are useless during the in-between cycles. In this case, a packet is broken down into 9 pieces and is sent progressively. Thus, higher throughput can be achieved, but with fewer decisions (crossbar configurations) between the same amount of data. In other words, a larger packet size is needed to fulfil higher throughput, but this can quickly become wasteful and inefficient.

In terms of hardware resources utilisation, as a direct comparison with GCQ [9], ignoring the switching performance, for a 256-bit 16x16 switch our S=4 implementation consumes around 3.5x the LUTs and 4x the FFs. It is important to note that it would be more appropriate for this comparison to be made against a competitor switch of wider packets for achieving similar throughput, as demonstrated below.

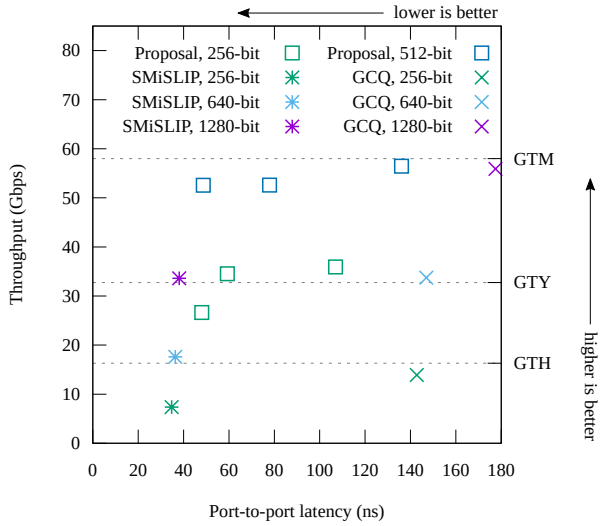

Figure 12: Comparison of 16×16 switch implementations on comparable FPGAs featuring the UltraScale+ architecture

Figure 12 shows how our implementation performs against the related work, based on the operating frequencies for VU9P [22]. For a packet size of 256 bits and S=2, our proposal achieves around 4.7 and 2.5 times the throughput of the 256-bit versions of SMiSLIP and GCQ respectively. SMiSLIP seems to marginally win for port-to-port latencies. However, since it uses the iSLIP scheduling algorithm, there will be significant additional latency for high input rates, as found in simulation. Moreover, if really needed, our design can easily achieve lower pipeline latencies by further exploring the S parameter. GCQ yields the worst latencies irrespective of packet size. For a packet size of 512 bits and S=1, we achieve the highest throughput overall, nearly saturating the UltraScale+ GTM transceiver. However, the best choice for GTM seems to be 512-bit packets and S=4 for our proposal. This is because for a small drop in throughput, it yields 3.7 times lower latency than our only competitor, a GCQ switch with more than twice the packet size.

7 FUTURE WORK AND CONCLUSION

As future work, it would be useful to explore other functionalities desirable to have in network switches. These include quality of service (QoS) support for prioritising different classes of traffic,

and the support for other addressing methods, such as multicast. The challenge in adding more functionality would be the resource utilisation efficiency, as it might be easy to incorporate additional features with more hardware resources. Our switch design was implemented out-of-context, but it could be tested on a high-end board, where multiple transceiver ports are populated.

The current presentation starting with a pipelined design and stages of similar slack simplified any discussions in relating it to different devices, architectures, and transceiver types. The alternative presentation to pipeline a combinational design could be particularly useful in targeting a specific device.

Note that today's switches are systems purposely built for the task and can provide links with a bandwidth of 400 Gbps [3], much higher than the the 58 Gbps supported on the latest FPGA device [14], although ports can also be combined [20]. Therefore, it would be interesting to study applications on larger and faster switches for newer technologies. For the moment, a comparison with proprietary technology could also focus on the algorithmic aspect of our approach, as we have shown that competing solutions can contribute to a higher port-to-port latency in all evaluated traffic patterns.

It would also be appropriate to build mathematical models estimating performance and memory needs, as well as proofs for any optimalities our design or algorithmic approach may exhibit.

This paper presents a novel high-performance network switch architecture for FPGAs. The main element is the *combined parallel round arbiter*, a fully-pipelinable structure with which packets are rearranged at line rate, capable of filling the output queues in balanced way. It can achieve a better switching performance using a fraction of the memory used by VOQs, as our simulations illustrate. Our approach is well suited for FPGA implementation, since it does not require crossbars with scheduling algorithms or any logic or memory speedups, leading to a high operating frequency. In combination with its output-per-cycle property, it eliminates the need for sacrificing the smaller packet size for achieving high throughput, as found in related work. We also present the register removal optimisation, with which the port-to-port latency is significantly reduced. Finally, a 16×16 switch is implemented on a Zynq UltraScale+ device, achieving an aggregate bandwidth of 903.2 Gbps, saturating its GTY transceivers. It almost also saturates the GTM transceivers found in the highest-end devices, while providing competitive switching performance for a wide range of traffic patterns.

ACKNOWLEDGMENTS

This research was sponsored by dunnhumby. The authors would like to thank Behrad Niazmand for his insightful feedback, as well as Chris Brooks and Rosie Prior from dunnhumby for their involvement in the partnership program. The support of Microsoft Research and the United Kingdom EPSRC (grant number EP/L016796/1, EP/I012036/1, EP/L00058X/1, EP/N031768/1 and EP/K034448/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653) is gratefully acknowledged.

REFERENCES

[1] Mohamed S Abdelfattah, Andrew Bitar, and Vaughn Betz. 2015. Take the highway: Design for embedded NoCs on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 98–107.
[2] Thomas E Anderson, Susan S Owicki, James B Saxe, and Charles P Thacker. 1993. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS)* 11, 4 (1993), 319–352.
[3] Arista Networks, Inc. 2019. 7800R3 Series Data Center Switch Router Data Sheet.
[4] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 307–314.
[5] Andrew Bitar, Jeffrey Cassidy, Natalie Enright Jerger, and Vaughn Betz. 2014. Efficient and programmable Ethernet switching with a NoC-enhanced FPGA. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 89–100.
[6] Daniel Bundala and Jakub Závodnỳ. 2014. Optimal sorting networks. In *International Conference on Language and Automata Theory and Applications*. Springer, 236–247.
[7] Jonathan Chao. 2000. Saturn: a terabit packet switch using dual round robin. *IEEE Communications Magazine* 38, 12 (2000), 78–84.
[8] Shang-Tse Chuang, Ashish Goel, Nick McKeown, and Balaji Prabhakar. 1999. Matching output queueing with a combined input/output-queued switch. *IEEE Journal on Selected Areas in Communications* 17, 6 (1999), 1030–1039.
[9] Zefu Dai and Jianwen Zhu. 2012. Saturating the transceiver bandwidth: Switch fabric design on FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 67–76.
[10] Nadeen Gebara, Jiuxi Meng, Wayne Luk, and Paolo Costa. 2018. Scheduling Algorithms for High Performance Network Switching on FPGAs: A Survey. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 166–173.
[11] Pankaj Gupta and Nick McKeown. 1999. Designing and implementing a fast crossbar scheduler. *IEEE micro* 19, 1 (1999), 20–28.
[12] Chunzhi He and Kwan L Yeung. 2011. D-LQF: An efficient distributed scheduling algorithm for input-queued switches. In *2011 IEEE International Conference on Communications (ICC)*. IEEE, 1–5.
[13] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.
[14] Xilinx Inc. 2015-2019. UltraScale+ FPGA Product Tables and Product Selection Guide.
[15] Xilinx Inc. 2019. Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics (DS923).
[16] Yossi Kanizo, David Hay, and Isaac Keslassy. 2009. The crosspoint-queued switch. In *IEEE INFOCOM 2009*. IEEE, 729–737.
[17] John Kim, William J Dally, Brian Towles, and Amit K Gupta. 2005. Microarchitecture of a high-radix router. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 420–431.
[18] Yihan Li, Shivendra Panwar, and H Jonathan Chao. 2001. On the performance of a dual round-robin switch. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Vol. 3. IEEE, 1688–1697.
[19] Yihan Li, Shivendra Panwar, and H Jonathan Chao. 2002. The dual round robin matching switch with exhaustive service. In *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*. IEEE, 58–63.
[20] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso and Sergio Lopez-Buedo. 2019. Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack. In *2019 30th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.
[21] Nick McKeown. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking* 2 (1999), 188–201.
[22] Jiuxi Meng, Nadeen Gebara, Ho-Cheung Ng, Paolo Costa, and Wayne Luk. 2019. Investigating the Feasibility of FPGA-based Network Switches. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE.
[23] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting networks on FPGAs. *The VLDB Journal—The International Journal on Very Large Data Bases* 21, 1 (2012), 1–23.
[24] M Hassan Najafi, David J Lilja, Marc D Riedel, and Kia Bazargan. 2018. Low-cost sorting network circuits using unary processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 8 (2018), 1471–1480.
[25] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2018. FLiMS: Fast Lightweight Merge Sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 78–85.
[26] Philippos Papaphilippou and Wayne Luk. 2018. Accelerating database systems using FPGAs: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 125–130.
[27] Philippos Papaphilippou, Holger Pirk, and Wayne Luk. 2019. Accelerating the merge phase of sort-merge join. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 100–105.