Hipernetch: High-Performance FPGA Network Switch

PHILIPPOS PAPAPHILIPPOU, JIUXI MENG, NADEEN GEBARA, and WAYNE LUK,
Imperial College London

We present Hipernetch, a novel FPGA-based design for performing high-bandwidth network switching. FPGAs have recently become more popular in data centers due to their promising capabilities for a wide range of applications. With the recent surge in transceiver bandwidth, they could further benefit the implementation and refinement of network switches used in data centers. Hipernetch replaces the crossbar with a "combined parallel round-robin arbiter". Unlike a crossbar, the combined parallel round-robin arbiter is easy to pipeline, and does not require centralised iterative scheduling algorithms that try 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. Our proposed Hipernetch architecture additionally provides a competitive switching performance approaching output-queued crossbar switches. Our implemented Hipernetch designs exhibit a throughput that exceeds 100 Gbps per port for switches of up to 16 ports, reaching an aggregate throughput of around 1.7 Tbps.

CCS Concepts: • **Networks** → **Bridges and switches**; **Packet scheduling**; **Packet-switching networks**; • **Hardware** → **Reconfigurable logic and FPGAs**; **Networking hardware**; **High-speed input/output**; **Hardware accelerators**; **Programmable interconnect**; • **Information systems** → **Data streams;**

Additional Key Words and Phrases: Network switch, FPGA, round-robin, arbiter, scheduling algorithms, sorting network applications, stream processing

ACM Reference format:
Philippos Papaphilippou, Jiuxi Meng, Nadeen Gebara, and Wayne Luk. 2021. Hipernetch: High-Performance FPGA Network Switch. *ACM Trans. Reconfigurable Technol. Syst.* 15, 1, Article 3 (November 2021), 31 pages.
https://doi.org/10.1145/3477054

1 INTRODUCTION

Although 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 [59, 60]. In the latest device iterations, the available logic resources have also increased substantially. For example, the relatively recently announced VU19P device features around 9 million logic cells.

This research was sponsored by dunnhumby, as well as Microsoft, Microsoft Research, Intel, and the United Kingdom EPSRC (grants EP/L016796/1, EP/N031768/1, EP/P010040/1, and EP/S030069/1).

Authors' address: P. Papaphilippou, J. Meng, N. Gebara, and W. Luk, Department of Computing, Imperial College London, UK; emails: {pp616, jiuxi.meng16, n.gebara17, w.luk}@imperial.ac.uk.

1936-7406/2021/11-ART3 $15.00
https://doi.org/10.1145/3477054

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-based network switch implementations are inspired by well-known approaches used for ASICs/ASSPs. Such approaches are usually classified by the position of their buffer memory. Output-queued network switches have been shown to perform better than input-queued switches [13]. 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 [18]. 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 [42], and with bigger flits, to achieve a moderately high throughput, inheriting the related limitations.

This article presents Hipernetch, a novel network switch designed to perform well as an FPGA-based implementation; it does not require iterative algorithms [42], logic speedup [18] or memory speedup. Unlike previous research on FPGA switches, we further consider the relationship between the memory utilization of our proposed design and its performance with different traffic patterns, showing that it is indeed competitive as a switching approach.

As a proof of concept, our *open source* implementation of a 16×16 switch provides an aggregate throughput of 1.7 Tbps on a Xilinx Alveo U280 FPGA, surpassing the last reported, theoretically achievable 819 Gbps [2], which requires an embedded network-on-chip [9].

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 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 or both traffic models and realistic data center workloads.
- Verilog generator script and simulators developed from scratch and open sourced to enable further research on the subject.[1]

This article is an extension of the paper "High-Performance Network Switch Architecture" [52], which appeared at the 28th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) 2020. The extended material mainly revolves around the analysis of the properties of the switch (Section 5), large-scale simulations demonstrating that unlike other architectures, our proposed solution approaches the performance of an output-queued crossbar switch in a realistic data center setting (Section 6.2), a thorough implementation evaluation on a Xilinx Alveo U280 board that achieves a higher throughput (Section 7.3), and a more comprehensive comparison with alternative approaches (Section 7.4) and commodity switches (Section 7.5).

2 BACKGROUND

2.1 Head-of-Line Blocking and VOQs

Head-of-line blocking is a performance bottleneck observed in input-queued crossbar switches and is known to limit the throughput to just 58.6% [38]. VOQs are widely accepted as a solution to head-of-line blocking. Rather than having a single queue at each input port, a queue is available for each output port to allow progress of packets destined to different output ports. This results in

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

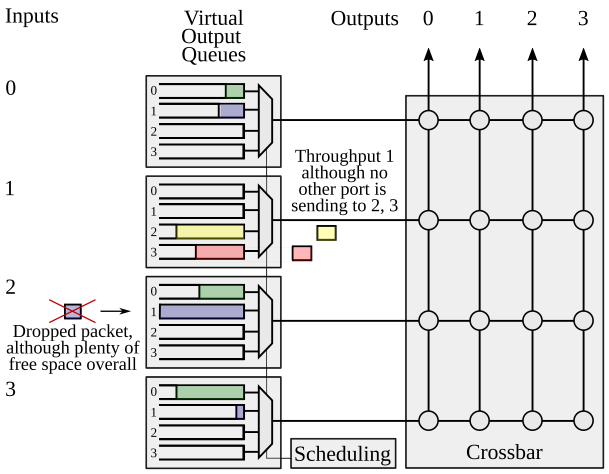

Inputs
Virtual Output Queues
Outputs 0 1 2 3
0
1
2
3
Throughput 1 although no other port is sending to 2, 3
Dropped packet, although plenty of free space overall
Scheduling
Crossbar

Fig. 1. An input-queued switch showing two limitations.

a total of $P \times P$ queues, where P is the number of switch ports. As demonstrated in Figure 1 (and in Section 6), VOQs have performance limitations. Such limitations include buffer fragmentation (the cause of the dropped packet in the example) and poor performance for bursty and nonuniform traffic (e.g., with the example where the multiplexer only allows either a yellow or an orange packet per cycle).

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 [7]. VOQs also exist in some form in hybrid memory model switches [18], such as in combined input and output queued (CIOQ) [13], combined input and crosspoint queued [27], and hierarchical crossbar [28] switches.

2.2 Packet Loss

Note that in contrast to other applications of interconnects, network switches can drop packets and are not responsible for complete transmission, such as by applying backpressure. Dropped packets are managed by the TCP/IP protocol, which sits at the network layer of the Open Systems Interconnection (OSI) model. If a software service uses TCP, any detected dropped packets are requested to be retransmitted. Network switches including our solution operate at the link layer level. However, minimising the packet loss rate is one of the motives for a higher network performance overall, as dropped packets can also involve packet retransmission.

2.3 Scheduling Algorithms

Scheduling or matching algorithms are used to determine which packets should be 'dequeued' and transmitted to output ports. Since crossbars only allow a single input-output port transmission, such algorithms typically attempt to resolve output port conflicts and make efficient input-output matches [21, 38]. This allows for maximising throughput while providing other desirable features, such as preventing starvation (i.e., ensuring no input is getting blocked indefinitely). The majority of scheduling algorithms for input-queued crossbar switches are iterative heuristics that 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 [38]. It is commonly applied to dequeue packets from 1 out of P queues, with a round-robin priority. The

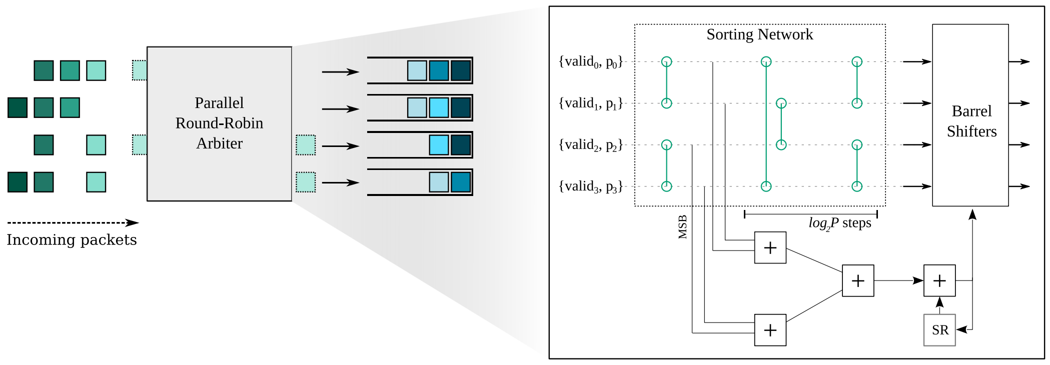


Fig. 2. Parallel round-robin arbiter functionality visualisation (left) and design abstraction (right), $P = 4$.

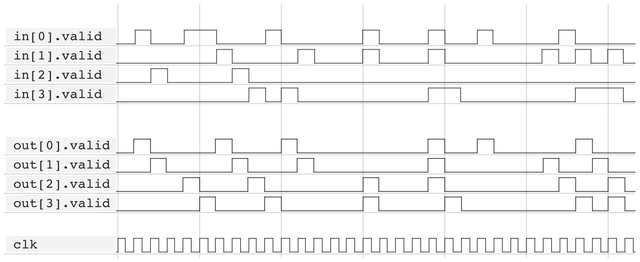


Fig. 3. Parallel round-robin arbiter waveform example, $P = 4$.

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), which can be expensive to implement efficiently and can contribute to the critical path for an increasing number of inputs [22]. 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, to improve scheduling performance for bursty traffic [34].

2.5 Parallel Round-Robin Arbiter

The parallel round-robin arbiter [53] (also known as *round-robin module*, not to be confused with that of Zheng et al. [62]) receives from 0 to P packets per cycle, and rearranges them such that P memory banks connected to its output appear to be written in a round-robin fashion. In contrast to a regular round-robin arbiter, which has a maximum output rate of 1, it is amenable to pipelining and provides a throughput of P packets per cycle.

Figure 2 provides a high-level description of the parallel round-robin arbiter and how it can support such a high throughput (left). It consists of non-blocking building blocks: a sorting network, responsible here for compacting the positions of the packets, an adder tree and barrel shifters to appropriately rotate the consecutive packets (see Section 4.2 for more details on those individual components).

As every packet is processed at line rate, there is no need for additional memory for producing the round robin effect. Figure 3 shows an example waveform of possible input and output of such an arbiter, capable of processing multiple requests at a time. The input involves packets arriving in arbitrary positions.

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 RELATED WORK

An FPGA network switch implementation is GCQ [18]. It improves on a hybrid buffer approach, the hierarchical crossbar switch [28]. 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, originally tested on an older device (Xilinx Virtex 6-240T).

A recent switch implementation on FPGA is SMiSLIP [42]. It is an input-queued switch, implemented as a crossbar with VOQs, and controlled by the iSLIP scheduling algorithm [38]. The focus of this work is the optimisation for FPGAs to achieve buffer sharing and 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 1,280-bit packets respectively on a Xilinx UltraScale+ device (VU9P).

This is expected for iterative matching algorithms, such as iSLIP. One of the original iSLIP-based prototypes [22] is an ASIC operating at 175 MHz in a 32-port switch. Despite its optimisations such as the merging of the two out of its three phases (request, grant and accept), it is not fully pipelined. It only produces output once every nine 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 nine 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.

4 HIPERNETCH: A NOVEL FPGA-BASED NETWORK SWITCH

The main idea of Hipernetch 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 VOQs, this requires $P \times P$ queues. However, by moving those queues towards the output, the traffic has the opportunity to get rearranged 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 4 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.

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.

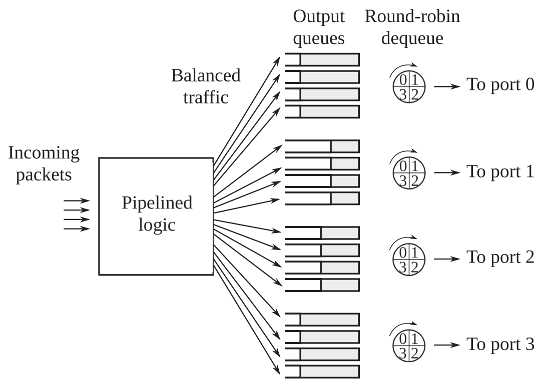


Fig. 4. Abstracted view of the proposed solution, $P = 4$.

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

4.1 Combined Parallel Round-Robin Arbiter

The main element in our solution is the *combined parallel round-robin arbiter* (Figure 5). 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 sorting network is used here to sort a batch of P incoming packets according to their validity and destination to ease routing. Essentially, by bringing packets with same destination together, the number of possible permutations required before forwarding them to the output ports is reduced to only include rotations. The 'offsets' pipeline includes everything below the sorting network in Figure 5, and mainly consists of adders, such as in the form of adder trees to count the packets going to each port, and a prefix sum that uses this information to find the starting position of each consecutive packet group. The two pipelines ('sorting network' and 'offsets') 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.

4.2 Building Blocks

Here, we provide an overview of the building blocks of Hipernetch, and how they interact with each other. Figure 6 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.

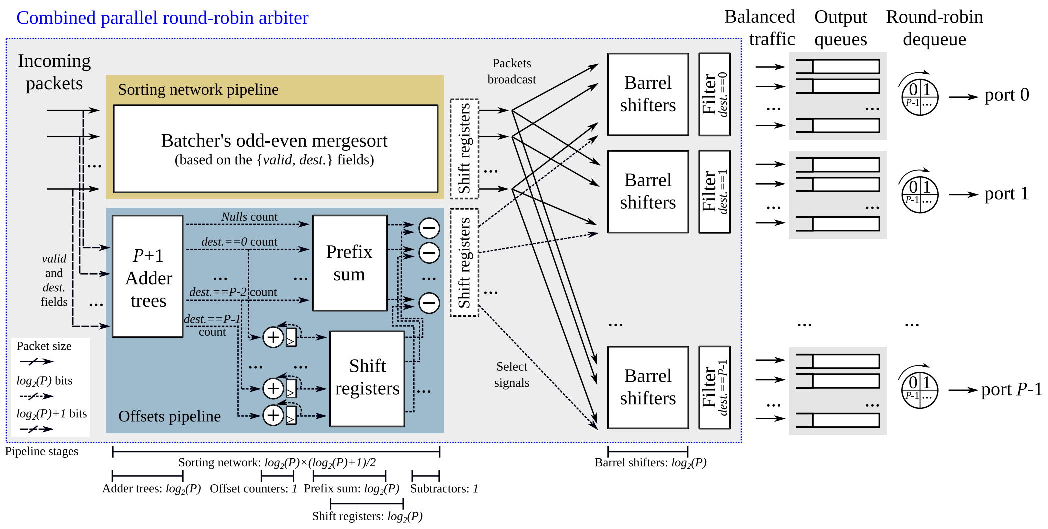


Fig. 5. The architecture of Hipernetch.

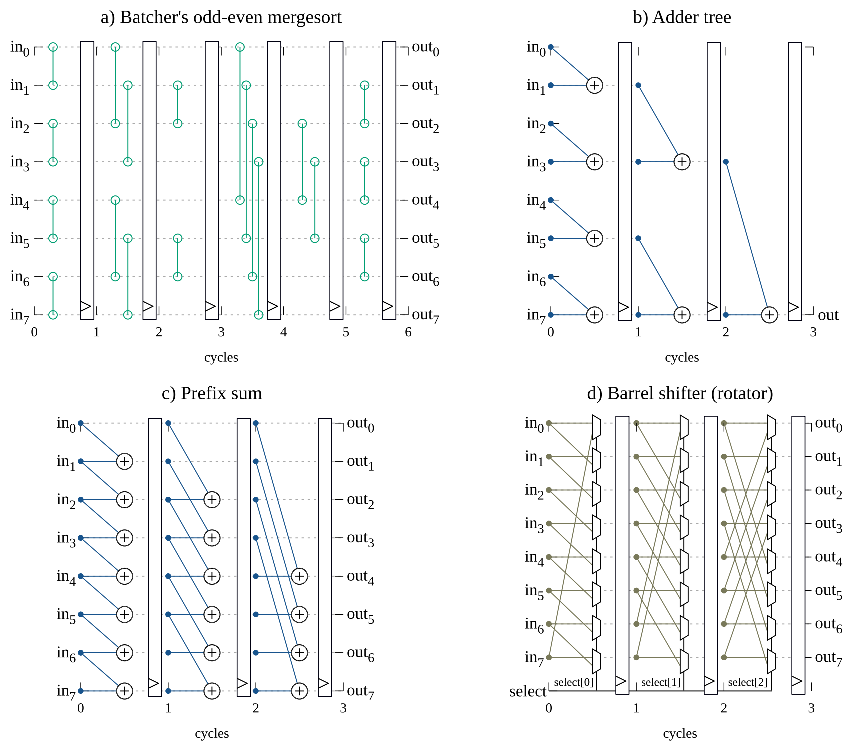

a) Batcher's odd-even mergesort
b) Adder tree
c) Prefix sum
d) Barrel shifter (rotator)
in0 in1 in2 in3 in4 in5 in6 in7
out0 out1 out2 out3 out4 out5 out6 out7
out
select
select[0] select[1] select[2]
cycles

Fig. 6. Pipelined versions of building blocks for eight inputs.

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 two elements. There are two topologies frequently used in FPGAs [45]: Batcher's odd-even merge sort and bitonic sort [8]. 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, although more optimal networks exist [10]. Depth-optimal sorters are difficult to find for large values of P. Figure 6(a) shows an example sorting network for eight inputs.

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.

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 6(b), 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—that is, 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).

Prefix sum. The parallel version of prefix sum [24] 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 $k \in \{0, 1, \ldots, P-1\}$) in $log_2(P)$ parallel steps, as in Figure 6(c).

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, to achieve the round-robin effect.

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 $port_0$).

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 6(d), 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.

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, and 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.

5 PROPERTIES AND OPTIMISATIONS

5.1 Complexity

In terms of hardware resource utilisation, the fastest growing of all Hipernetch 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 ($latency_{offsets}$).

$$\begin{aligned} latency &= latency_{BS} + max\ (latency_{SN},\ latency_{offsets}) \\ &= log_2(P) + max\ \left(\frac{log_2(P) \times (log_2(P)+1)}{2},\ 2 \times log_2(P) + 1\right) \\ &\in O(log^2(P)) \end{aligned} \quad (1)$$

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.

5.2 Reducing the Port-to-Port Latency

The logic complexity of each stage, although highly parallel, 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 7.

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 5. 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).

$$index_{comp} = log_2(P) \quad (2)$$

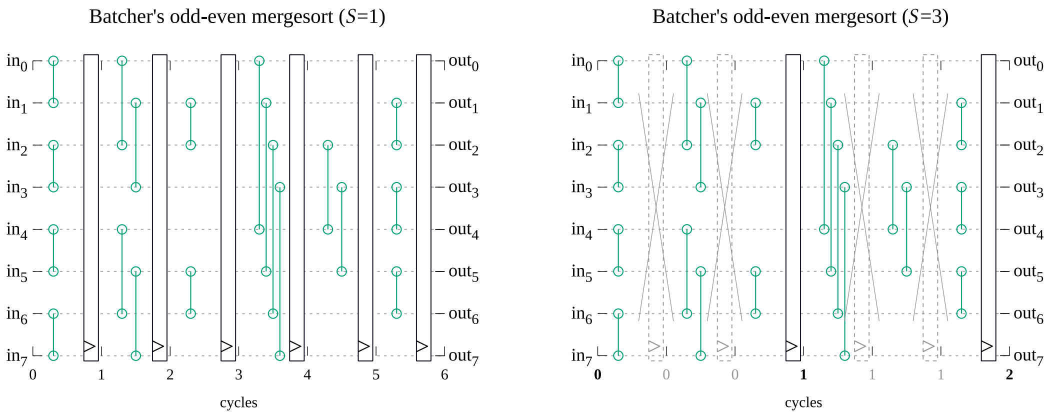

Batcher's odd-even mergesort (S=1)
Batcher's odd-even mergesort (S=3)
in0
in1
in2
in3
in4
in5
in6
in7
out0
out1
out2
out3
out4
out5
out6
out7
cycles

Fig. 7. Dividing the latency of a sorter by 3, before (left) and after (right).

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

$$latency_{opt} = \left\lceil \frac{latency - index_{comp}}{S} \right\rceil + \left\lceil \frac{1 + index_{comp}}{S} \right\rceil - 1 \quad (3)$$

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 [45] and (c) with *unary processing* [46], 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.

Table 1 illustrates how this optimisation significantly reduces the port-to-port latency, after also considering the optimisation of the next section. Section 7 contains more details about the impact of this optimisation on **flip-flop register (FF)** utilisation, as well as on operating frequency. Since the latency here represents only the minimum processing latency, in simulations we measure the average time a packet stays in the queues using different approaches and traffic patterns.

5.3 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 $P > 8$.

This also moves the placement of the compulsory register. In future work (Section 8), there is also a discussion for further exploration the placement of the compulsory register position. The compulsory register index is now $log_2(P)$ stages before the the common ending of the sorting

Table 1. Indicative Port-to-Port Latency after Reg. Reduction

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	5	9	14	20	27
Pipeline latency, $S = 1$	4	7	10	14	20	27	35	44
Pipeline latency, $S = 2$	2	4	5	7	10	14	18	22
Pipeline latency, $S = 4$	1	2	2	4	5	7	9	11
Pipeline latency, $S = 8$	1	1	1	2	3	3	4	6
Port-to-port latency (ns) (*indicative*, with $S = 4$ and $f_{clk} = 150$ MHz)	13.3	20	20	33.3	40	53.3	66.6	80

network and offsets pipeline. This is calculated as follows.

$$\begin{aligned} index'_{comp} &= max\ (latency_{SN},\ latency_{offsets}) - 1 - log_2(P) \\ &= max\ \left(\frac{log_2(P) \times (log_2(P) + 1)}{2},\ 2 \times log_2(P) + 1\right) - 1 - log_2(P) \end{aligned} \tag{4}$$

Table 1 presents the indicative results for the port-to-port latency, based on the optimised pipeline latency. Illustratively, one more cycle is added to the pipeline latency, representing the minimum time a packet stays in the output queues. A typical operating frequency of 150 MHz was selected to enable translating the pipeline latency from cycles to nanoseconds, before the actual place-and-route in the evaluation (Section 7).

5.4 Simpler Output Arbiters

There are alternative options for the output arbiters, instead of using traditional *round robin arbiters*. Although applicable for approaching OQ-xbar performance, 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.

The output arbiters can be simpler for the proposed solution. 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 $P - 1$).

This optimisation assumes no packets are being dropped; otherwise, when the FIFO space is not enough, there is a performance overhead, as the output arbiter can get out of synchronisation. In our single-switch simulations, the desynchronisation effect is taken into account, and the performance overhead was found to be negligible for the studied traffic patterns and performance metrics. When the output arbiter gets desynchronised, a few packets can wait in a port's FIFO group until the first row is filled, which is generally unwanted. There are some small workarounds suggestions when needed: (a) counting the modulo of the amount of dropped packets per port and accordingly skipping empty spaces when available, (b) when there is at least one dropped packet, delete the entire last row of P packets to avoid desynchronisation altogether (c) implement backpressure (see end of Section 8), but for dropping the packets before the pipeline, as means to keep the offset counters synchronised, and (d) use regular round-robin arbiters (with priority encoders), especially when the implementation on the target architecture is efficient enough.

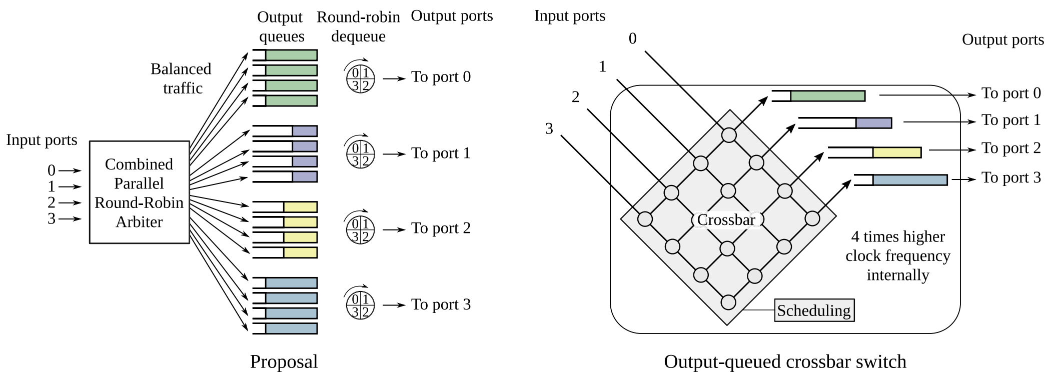


Fig. 8. Comparing the proposal to an output-queued crossbar-based switch, $P = 4$.

5.5 Resemblance to Output-Queued Crossbar-Based Switches

Output-queued crossbar switches (OQ-xbar) are considered as the ideal switch architecture and most alternative architectures strive to emulate their performance [13, 36, 44]. They consist of a crossbar fabric and a FIFO-based queue for each output port. In crossbars, conflicts happen when packets from different input ports are targeting the same output port at the same cycle. To solve this problem, both the crossbar and the queues use P times higher clock frequency than the base clock (fabric speedup of P times, where P is the number of ports). This gives the illusion that packets arriving at the input ports are forwarded "immediately" to the appropriate destination port queue as can be seen in the right-hand side of Figure 8.

In practice, the main performance bottleneck of the output-queued crossbar switch is the need for a high memory accessing speed. Alternatives have been researched to match the performance of an output queued switch. These solutions, however, are all based on the speedup of the switch fabric. In the work of McKeown et al. [39], a speedup of 2 in a CIOQ switch was proven to be sufficient to emulate an output-queued switch in theory. However, the need for scheduling algorithms that run with a speedup of 2 for switches with high port rates is challenging in general, and even more challenging on FPGAs due to their low operating frequencies. In this section, we demonstrate that our proposed design approaches the performance of the output-queued crossbar, although it does not necessarily behave identically to an OQ-xbar switch. More specifically, two packets arriving on the same cycle with the same destination can appear in different order at the output port, but this does not affect the common performance metrics.

In high level, an OQ-xbar switch with P ports can enqueue up to P packets to a single output port buffer immediately. The order at which these packets are written to the same destination port buffer depends on an arbitration policy used, which can be as simple as a round-robin arbitration policy. Our proposed switch in contrast relies on a sorting network to perform this arbitration and therefore provides no deterministic order in which packets are written to the destination port buffer. This is due to the fact that sorting networks are not stable sorters. Another reason the behaviour of our design is not exactly identical to that of an OQ-xbar is that packet drops can cause our optimised output arbiters to get desynchronised. In the rest of this section, we explain how these differences in behaviour impact the performance of our proposed design, and how it compares to an OQ-xbar in terms of the average latency, worst-case latency, throughput and packet loss rate.

Latency. Two metrics for measuring the switching performance are the *average* and the *worst-case* latency. The average and worst-case latency are measured by counting the average and maximum number of cycles respectively a packet needs to be served from the input port to the output port.

Without loss of generality, we study the events in a single output port out of the P available. With respect to the latency of packets destined to the same output port, there are two cases to consider: (1.) packets arriving on different cycles and (2.) multiple packets arriving on the same cycle.

For (1.), any two packets A and B arrive on different cycles t_A, t_B for the arrival time of A and B, respectively, where $t_A < t_B$ (B precedes A). Both the *combined parallel round-robin arbiter* and the output-queued crossbar have a fixed processing latency. Thus, it is guaranteed that the order of packets A and B will be conserved (each being served on the arrival time plus the processing latency) and (1.) is not a differentiating factor.

For (2.), multiple packets (up to P packets) arriving on the same cycle, it is shown that different permutations do not affect the latency metrics. Let $t_1, t_2, \ldots, t_p$ be the timestamps for the $i \in (1, 2, \ldots, P)$ consecutive packets to be dequeued. Note that these timestamps are one cycle apart from each other, due to the round-robin arbiters in the end. t_0 denotes the common timestamp of arrival in the input. The worst-case (t_{worst}) and average (t_{avg}) latencies are given by the following.

$$\begin{aligned} t_{worst} &= max((t_1 - t_0), (t_2 - t_0), \ldots, (t_i - t_0)) \\ &= max(t_1, t_2, \ldots, t_i) - t_0 \end{aligned} \tag{5}$$

$$\begin{aligned} t_{avg} &= ((t_1 - t_0) + (t_2 - t_0) + \cdots + (t_i - t_0))/i \\ &= (t_1 + t_2 + t_3 + \cdots + t_i)/i - t_0 \end{aligned} \tag{6}$$

Due to the associative and commutative property of both the max operation and summation, the queuing order at the output will not affect the worst-case or average latency for both our design and the OQ-xbar.

As the average latency is shown to be identical, we can verify this finding by comparing each set of output queues to an M/D/1 model, for example, as it is shown to model the OQ-crossbar as well [3, 63].

In our design, the highest arrival rate for queue group in the output is one packet per cycle (corresponding to 100% input rate under uniform Bernoulli traffic), and the highest serving rate is also one packet per cycle per port. The queue utilisation and is given by $\rho = \frac{\lambda}{\mu}$, where λ (or r, ranging from 0 to 1) is the arrival rate and μ is the serving rate. For instance, achieving the steady state of the model requires $\lambda \leq \mu$; otherwise, the queue size would grow indefinitely.

As the average latency metric comparison assumes infinite queues, we choose to set μ to 1 due to the dequeue policy of processing one packet per cycle (as with the OQ-xbar). The average latency ω_Q of a packet in such an M/D/1 queue can then be given by the following.

$$\omega_Q = \frac{\rho}{2\mu(1-\rho)} = \frac{\frac{\lambda}{\mu}}{2\mu(1-\frac{\lambda}{\mu})} = \frac{\lambda}{2(1-\lambda)} \tag{7}$$

By mapping the average latency to the y-axis and the input rate to the x-axis, we can compare this model to our simulation results. This simulation dataset represents the average latency for the entire switch (including the queues for all output ports), using uniform Bernoulli traffic. There is a minor displacement due to a convention to wait to empty the queues, marginally affecting the effective input rate in the simulation (see Section 6 for more details about this simulation). The average latency ω_Q of multiple M/D/1 queues equals to the average latency of a single M/D/1. Thus, observing this juxtaposition in Figure 9 indicates that our switch design follows the output-queuing model.

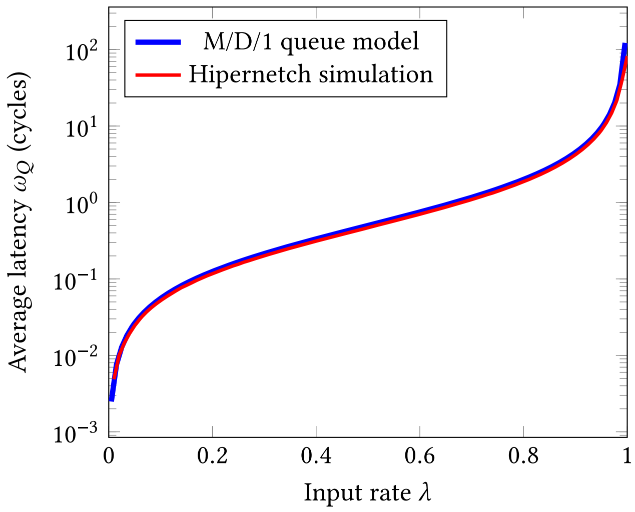


Fig. 9. Average latency comparison of the M/D/1 queueing model and simulation result.

Throughput. The throughput is measured as the number of exported packets for a given time period. Both the output-queued crossbar fabric and the *combined parallel round-robin arbiter* are collision free—that is, given at most P packets targeting the same port, arriving on the same cycle, where P is the number of ports, they can all be processed. In the case of the crossbar, this is achieved by expanding the time domain (i.e., speeding up both the fabric and the queues P times). In the case of our design, this is achieved by rearranging the packets in line rate using non-blocking components and increasing the parallelism of the space domain (i.e., accessing the FIFOs in banks). They both support the line rate, sometimes referred to as "supporting 100% of the throughput", although this holds as long as there is adequate free space in the queues. As network switches do not have an infinite amount of queue space, the throughput is reduced by packet losses.

Packet loss rate. In terms of the packet loss rate, our proposal performs similarly to OQ-xbar with queues of similar total size. As this metric relates to a fixed queue size, the implementation of the output arbiter and desynchronisation workaround also affects the packet loss behaviour. For brevity, we show that our proposal with any simpler output arbiter choice is no worse than an OQ-xbar with $(M - P)/M$ of the total queue space for a port, where $M = P * D$ is the space of an output queue group and D is the queue depth. Note that this is a worst case, and also between the workarounds there is, for instance, the backpressure solution which would perform the same as an OQ-xbar with the same total queue size. The longer the FIFOs, the less important this difference becomes. This is also apparent in simulation results, such as in Figure 13 of Section 6.1, where such small variation by 1 in the FIFO depth (y-axis) becomes less important for packet loss at higher values.

On each cycle, due to the round-robin enqueue and dequeue policies in our solution, we notice that the available spaces will be consecutive (cyclic) and evenly distributed across the P output queues of the port. Therefore, regardless if the output arbiter for the port is synchronised or not, when there are P or more free spaces ($F \in P, P + 1, \ldots, P \times D$), all packets arriving on that cycle will find a place in the FIFO group of the port and will not be dropped. Therefore, we notice that it performs at least better than an OQ-xbar with $(M - P)/M$ the queue space (with P less spaces).

5.6 Output Per Cycle

The *combined parallel round-robin arbiter* produces up to P packets in every FPGA cycle, without depending on algorithm phases or iterations. In addition, as with modern designs for high-throughput data processing [48, 51], 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 [22, 42]. According to how the workload is partitioned across the clock edges, the operating frequency could be much higher and still underperforming in terms of throughput. Essentially, our solution can support many times more throughput for the same packet size than iterative approaches. See the related work at Section 3 for a more complete discussion on the shortcomings of iterative scheduling algorithms, and Section 7.4 for numerical examples based on implementation.

6 SIMULATION-BASED EVALUATION

We implement our proposed Hipernetch switch architecture in software and conduct high-level (non-RTL) simulations to evaluate the performance of our proposed switching algorithm. Hardware intricacies such as the pipeline latency, bandwidth and packet size are not considered, and we assume that all considered switching algorithms run on a switch at the same operating frequency/ bandwidth and with a 0 processing latency. The main purpose of this approach is to focus on the algorithmic aspect of our proposal. We first evaluate the algorithm in a single switch using a variety of traffic patterns and performance metrics associated with a switch (Section 6.1). Then, we evaluate the performance of our proposed switch in a more realistic multi-tier data center topology using common data center performance metrics and realistic traffic traces (Section 6.2).

6.1 Single Switch Algorithm Evaluation

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 [42], outperforming a solution with a hybrid memory model [18].

The performance of the proposed design is compared against various scheduling algorithms. We present comparison results for **parallel iterative matching (PIM)** [6], **round-robin matching (RRM)** [38], iSLIP [38], **dual round-robin matching (DRRM)** [12, 33] and **dual round-robin matching switch with exhaustive service (EDRRM)** [34].

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 [33, 38], scheduling performance and/or their behaviour under a wider spectrum of traffic patterns [23, 34].

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 $1/P$.

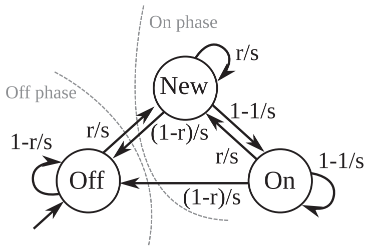


Fig. 10. A uniform bursty traffic Markov model.

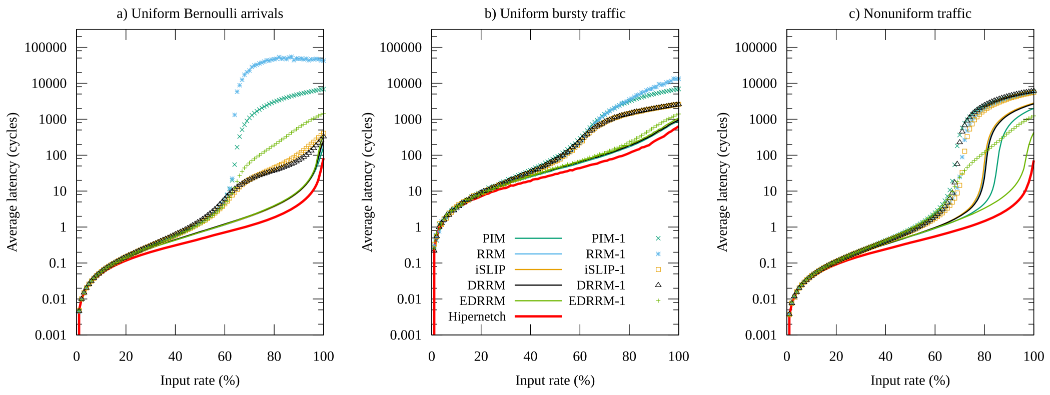


Fig. 11. Average packet latency using different algorithms and traffic models (16 × 16 switch, simulation).

(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 10, 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 [23], but it does not limit the maximum input rate to $1 - \frac{1}{s+1}$. The default value of s is set to 32.

(c) *Nonuniform traffic* [23, 34]: 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 11, 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 [34], which is selected here as an algorithm designed for bursty/nonuniform traffic. Still, for an input rate of near 1, our proposed approach reduces the average latency by 2×, 1.4× and 6× for the traffic models (a), (b) and (c) over the best alternative for each (PIM, PIM and EDRRM).

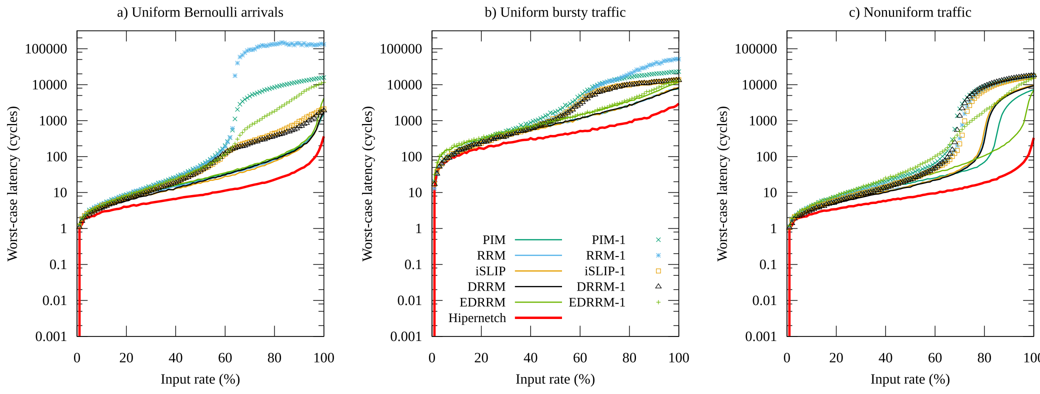


Fig. 12. Average worst-case packet latency using different algorithms and traffic models (16 × 16 switch, simulation).

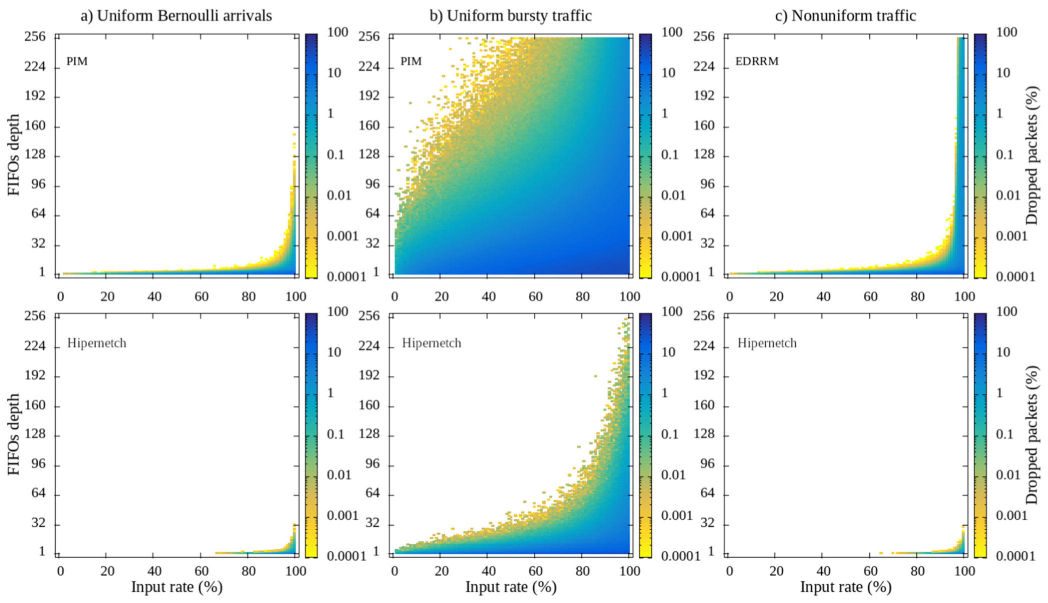


Fig. 13. Impact of queue size, for the proposal and the best alternative per traffic model (16 × 16 switch).

Worst-case latency. Using a similar experiment, for a 16 × 16 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 near 100%, the best alternative per traffic model yields a latency of 1,535, 7,992 and 7,212 cycles for the traffic patterns (a), (b) and (c). The corresponding values for our proposal are 375, 2,807 and 310 cycles, which translate to a reduction of the worst-case latency by 4.1×, 2.8× and 23.3× on average, respectively. Figure 12 summarises the worst-case latency simulation results.

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 FIFO lengths. The FIFO

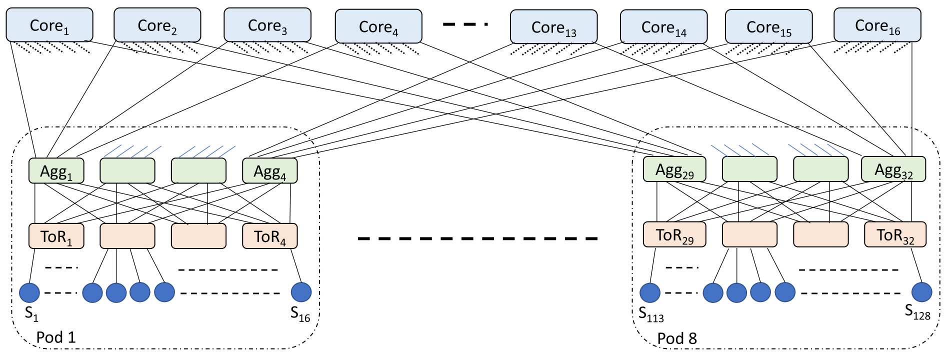


Fig. 14. Simulated Network Topology.

depth metric is a consistent way to measure the total buffer size in our simulations, as both VOQs and our proposal's output queues are $P \times P$ buffers.

Figure 13 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—that is, a dropped packet rate equal to 0 (represented by the colour white). For the traffic pattern (a) and an input rate near to 1, we achieve a 4.6× reduction in FIFO depth (153 to 33) for supporting 100% throughput, whereas 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 close to 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. Moreover, for the traffic pattern (c), it is able to handle nonuniform traffic with marginally less requirements than in the Bernoulli case, whereas this pattern becomes much more challenging for the other approaches.

6.2 Large-Scale Data Center Simulations

In this section, we evaluate the performance of our proposed switch architecture in a more realistic data center setting by performing packet-level simulations using the discrete event simulator Omnet++ [1]. We only consider the best alternative scheduling algorithms identified in the previous section (PIM and EDRRM), and further consider a switch with an output-queued crossbar fabric (OQ-xbar) to serve as the ideal baseline. Our simulation methodology and performance results are further described next.

Methodology. We consider a common three-tier network topology described in the work of Al-Fares et al. [4] and commonly used in data centers. As depicted in Figure 14, the topology consists of switches with a fixed number of ports (k), and has a regular architecture parametrised by k which is also equal to the number of pods or server groups. Specifically, we simulate a topology with k = 8 implying a total of eight pods. Each pod has 16 servers (nodes) that are interconnected through 4 Top of Rack switches (ToR) and Aggregation switches (Agg). The pods are connected to each other through 16 core switches connected to Aggregation Switches in the pods. All network links and switch ports in our simulation operate at a line rate of 10 Gbps. We further assume that link propagation delays are negligible and therefore do not model them in our simulations.

The end host servers attached to the ToR switches run applications that use a TCP congestion control protocol with selective acknowledgement (TCP SACK) implying that lost packets are identified and re-transmitted [37]. The starting window size is set to 2 packets, and the slow start threshold is set to 64 packets. We assume that packet corruption is unlikely, and the only trigger for packet loss is switch buffer overflow. Specifically, each switch port is assigned a fixed buffering capacity (4 MBytes), and new packets are immediately dropped if they result in exceeding the port's buffering capacity (tail drop). For switches with VOQs, the port buffer capacity is equally divided across all VOQs at a port. Switches route packets across servers using the Equal Cost Multipath Protocol (ECMP) routing protocol [26].

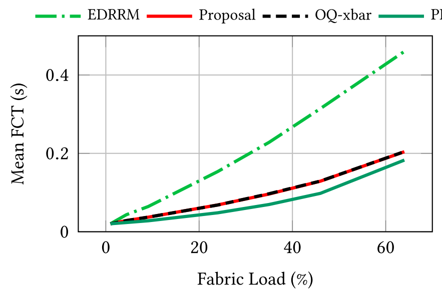
We evaluate the performance of the various switch architectures by relying on empirical traffic distributions that are representative of workloads observed in real production data centers described in the work of Alizadeh et al. [5]. Specifically, a web search workload and a data mining workload are considered. The term flow is used to describe a series of packets belonging to a specific application and having a specific source node and a destination node. Unlike the rest of the article, where we assume packets consisting of one flit, we assume a fixed packet size of 4,032 Bytes throughout the simulation. We opt for a large packet size to avoid excessive simulation runtime duration. Nonetheless, we ensure that the packet size is less than the smallest flow size to avoid minimise overhead when transmitting short flows. It is worth noting that given our simulation assumptions, the packet size does not impact the obtained performance results.

Since only the flow size distribution of those workloads is available, such as in previous research [5], the source and destination nodes of the flows are chosen out of the 128 available servers with a uniform probability distribution. Additionally, flows belonging to both workloads are generated based on the provided workloads' flow size distribution and arrive according to a Poisson process. The inter-arrival rates of both flow distributions are jointly varied to investigate the performance of the switch architectures under various fabric loads, where the fabric load is representative of the percentage mean link utilisation.

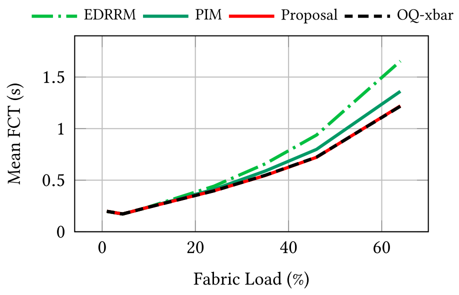
The *flow completion time (FCT)* is used as the main metric for evaluating our proposed switch architecture under the assumption that traffic flows have no deadlines. The flow completion time is the time taken for a flow to be received and acknowledged by a receiving node. The flows of both workloads are divided into long flow and short flow buckets based on their sizes. Flows with fewer than 20 MBytes are considered as short, whereas those exceeding 20 MBytes are considered to be long. The average flow completion time is reported for both buckets, whereas the 99th percentile flow completion time is only reported for short flows. We further consider the *packet loss rate* and *mean queuing time* observed in switches, and are shown in Figure 18. The packet loss rate represents the ratio of packets lost across all switches with respect to the total packets sent by servers, whereas the mean queuing time is the average of the time packets spend in switches queues.

Long flow completion time. The difference in the distribution of flow sizes between the web search and data mining workloads is evident in their mean flow completion times shown in Figure 15. As can be seen, the data mining workload has significantly longer flows resulting in both longer flow completion time and a smaller performance gap between the EDRRM switch and other switch architectures considered. The exhaustive policy of EDRRM is less favourable towards the web search workload as the flow completion time of long web search flows is 2× higher when using an EDRRM switch as opposed to other considered architectures. Our proposed architecture performs very closely to an output-queued crossbar switch as explained earlier. Overall, it can be seen that the PIM switch also provides a comparable flow completion time to that of our proposed architecture.

Short flow completion time. Like the case of long flows, our proposed architecture also results in completion times that are equivalent to those observed when using an output-queued crossbar switch. The PIM switch provides a slightly lower flow completion time compared to the output-queued crossbar switch for both the web search and data mining workloads shown in

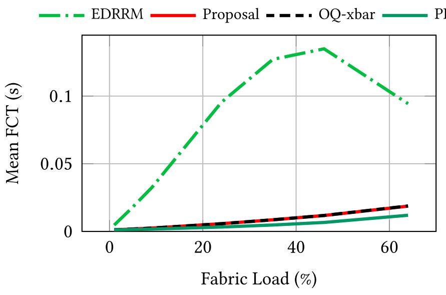


(a) Web search workload

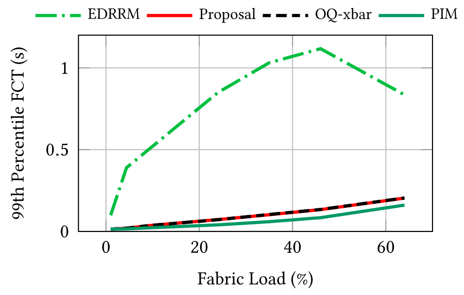


(b) Data mining workload

Fig. 15. FCT results of long flows.



(a) Mean FCT

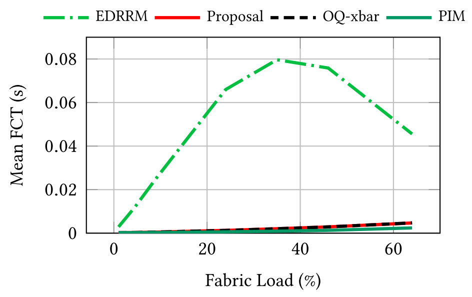


(b) 99th Percentile FCT

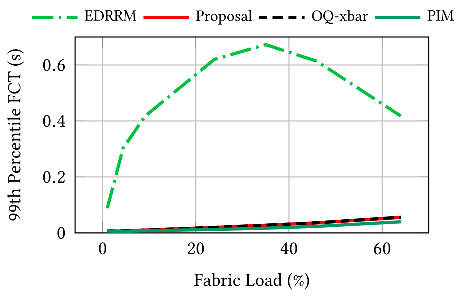
Fig. 16. FCT results of short web search flows.

Figures 16 and 17, respectively. This occurs since the input-buffered VOQ architecture has a lower effective buffering capacity than an output-queued crossbar switch. As a result, PIM switches start observing packet losses sooner than output-queued switches as well as a smaller mean packet queuing time. Since the TCP protocol reacts to packet losses, the packet loss rate does not increase with a higher fabric load. Since the overhead of packet re-transmission is negligible compared to the longer queuing times of output-queued crossbar switches shown in Figure 19, PIM switches can therefore provide short completion times when coupled with TCP. Alternative transport protocols that react to queue buildup and not only packet loss can help improve the performance of output-queued crossbar switches compared to PIM switches; however, this is beyond the scope of our work. EDRRM switches in contrast observe both queue buildup as well as packet loss as a result of the inefficiency of the scheduling policy. Specifically, the exhaustive policy results in starving the shorter flows, and as a result the 99th percentile flow completion time of the EDRRM switch is 13× and 5× higher than alternative architectures for the data mining workload and web search workload, respectively, and the mean flow completion time is approximately 20× and 7× higher for the data mining workload and web search workload, respectively.

In conclusion, this section investigated the performance of the various scheduling algorithms in a realistic large-scale data center setting. Unlike the stand-alone experiments, this section considered the time it takes to complete flow transfers and not only the time for switching packets. Our results show that exhaustive scheduling strategies such as EDRRM are actually poor candidates in realistic settings that consider flow completion time as their main performance metric. Further, they support the fact that our proposed solution matches the performance of an OQ-Xbar switch.



(a) Mean FCT



(b) 99th Percentile FCT

Fig. 17. FCT results of short data mining flows.

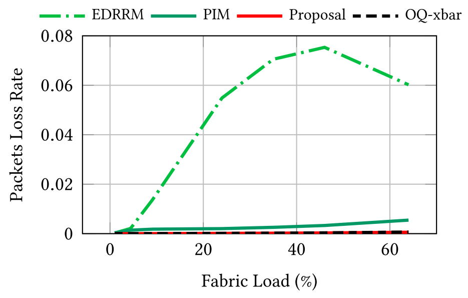


Fig. 18. Packet loss rate.

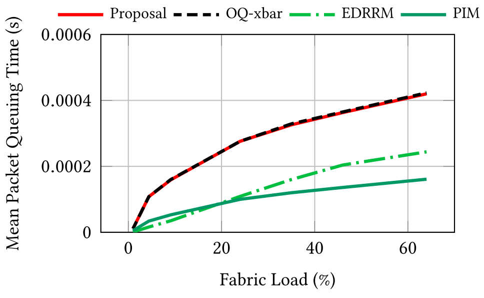


Fig. 19. Mean packet queuing time.

7 FPGA-BASED EVALUATION

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

To restrict our design space and focus on the more demanding switch logic, the FIFO queue depth is limited to 1, implemented as registers. The effects of different memory technologies and hierarchies for different devices and number of ports are beyond the scope of this work. It is worth noting that 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 [42] (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.

The design space for this experiment was switch implementations for 256-bit, as well as 512-bit packets, the number of ports ranging from 2 to 32 (powers of 2) and the latency reduction parameter S in the range from 1 to 8.

7.1 Validation

We implement a parametrisable RTL generator script that produces Verilog code implementing our proposed switch design given a specified packet size, number of ports (P) and latency reduction (S) requirement. 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, as well as on Xilinx Alveo U280 through JTAG-based debugging. In the latter case (b), the generated Verilog code is encapsulated in an AXI peripheral with the following debugging functionality: a batch of up to P packets is stored

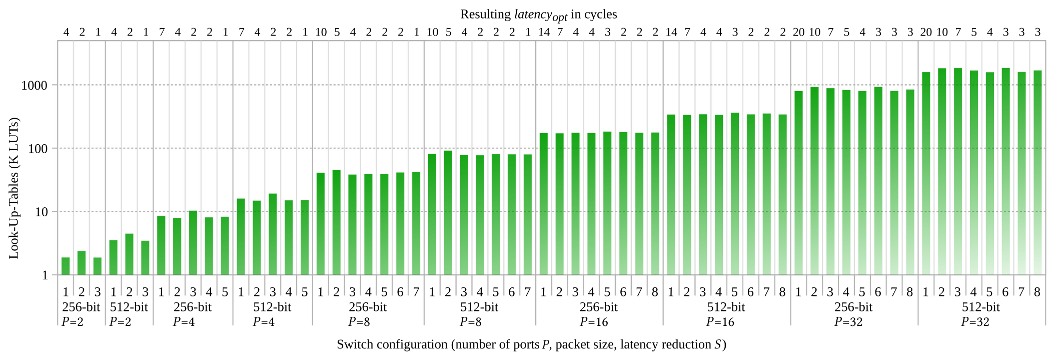


Fig. 20. LUT utilisation.

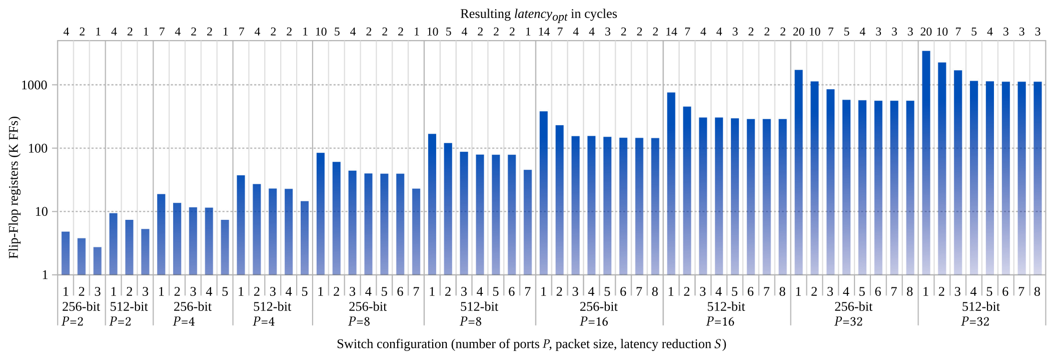


Fig. 21. FF utilisation.

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. The retrieved packets are then compared against the expected packets (corresponding to the non-RTL simulations).

7.2 Resource Utilisation

First, the resource utilisation in FFs and **lookup tables (LUTs)** of Hipernetch is presented, as reported by Vivado 2020.1.

Figure 20 shows the result of this exploration in thousands of LUTs (K LUTs). As an observation, the number of ports is a big contributor to the LUT utilisation and could become a limiting factor for big switches. Additionally, the S parameter for latency reduction does not seem to have a big impact on the LUT utilisation. This is expected as the value of S only determines the number of pipeline registers and does not impact the rest of the logic of the switch.

In contrast, Figure 21 illustrates the flip-flop utilisation for the switch designs in the design space. The number of registers can be significantly decreased by the register reduction optimisation (S approximately divides the pipeline stages). As observed, the synthesis results heavily reflect the choices for the S parameter. For instance, for $P = 16$ and 512-bit packets, if we use approximately one quarter of the pipeline registers (using $S = 4$, yielding around 3.5× stages reduction), the FF utilisation drops from 755K to 305K (around 2.5 times FF reduction).

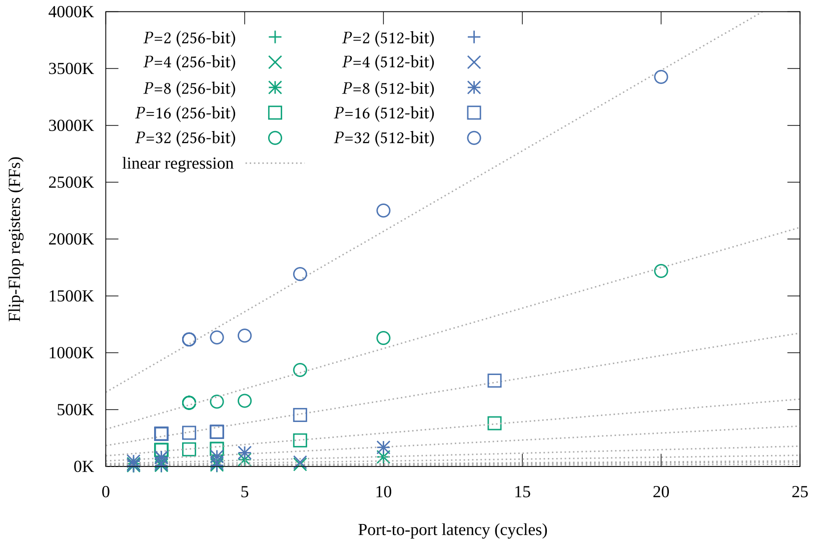


Fig. 22. Correlation between FF utilisation and $latency_{opt}$.

In both Figures 20 and 21, summarising the LUT and FF synthesis results respectively, there is a secondary x-axis. The ticks in this axis represent the optimised port-to-port latency ($latency_{opt}$) values, or in other words the final number of pipeline stages or registers for the corresponding design. One observation is that there are different values of S for the same P that map to the same $latency_{opt}$ and this correctly follows the respective formula. However, those designs with the same number of pipeline registers are not equivalent, as their pipeline registers can be in different placements. In contrast, when S is sufficiently large, the $latency_{opt}$ becomes equal to 1, in which case the only register remaining is the compulsory register. In the latter case, the designs with a single pipeline register are equivalent, as the compulsory register position is a function of P. For these reason, the redundant data points have been removed from the figures, hence the points for $P = 2$ stopping at $S = 3$, for example.

The correlation between the amount of FFs and the pipeline latency is illustrated more explicitly in Figure 22, that features modelling and non-logarithmic axes. Each one of the series, containing the FF utilisation for different values of S, is mapped to the corresponding $latency_{opt}$ and fits a simple linear regression model. With some small variation, this relationship seems more or less linear.

7.3 Performance

The performance results for this device are summarised in Figure 23. As a target device, we select the Alveo U280 based on the Xilinx UltraScale+ architecture. There are 256-bit (left) and 512-bit-wide (right) versions of 2, 4, 8 and 16-port switches. Each of those eight data series has multiple points, one for each S value in $\{1, \ldots, 8\}$ (with identical points removed). 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 secondary y-axis shows the f_{max} for all points inside each plot.

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., $latency_{p2p} = (latency_{opt} + 1)/f_{clk}$).

As shown in Figure 23, the plot for the 512-bit-wide design space (right), there is at least one configuration that exceeds 100 Gbps for each explored number of ports ($P = 2, 4, 8, 16$). For

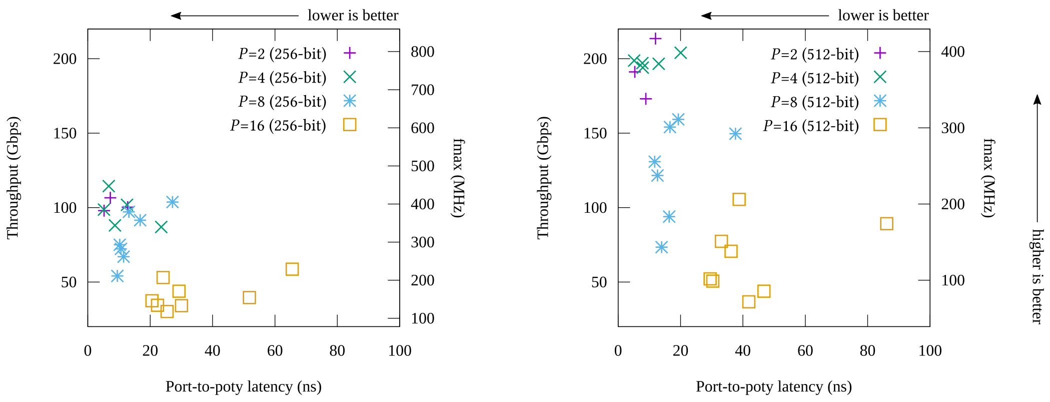


Fig. 23. Throughput and port-to-port latency on Alveo U280.

256-bit packets (left), it is more challenging to get high throughput for a similar range of operating frequencies, but there are still points exceeding 100 Gbps for ports 2,4 and 8, but not for $P = 16$. For $P = 16$, one good-performing configuration from the design space is for $S = 2$ and 512-bit, which achieves an aggregate throughput of 1.7 Tbps and a port-to-port latency of 39 ns. Note that this specific data point was applied aggressive optimisation, as explained in the following paragraph, suggesting that further improvements could be made by exploring tool optimisations.

Not shown in Figure 23 are the 32-port switch configurations, out of which only two were successfully implemented: 256-bit switches with $S = 3$ and $S = 5$. The achieved port-to-port latency was 166 and 129 ns, and the port throughput 12.3 and 10 Gbps respectively for $S = 3$ and $S = 5$. Those implementations used special Vivado directives (e.g., AlternateRoutability, ExploreSequentialArea, SSI_SpreadLogic_high, SSI_SpreadSSLs or AggressiveExplore) to overcome some routing complications from using multiple Super Logic Regions. Although modular, the switch design requires frequent communication between modules. This becomes less of a problem when most logic fits inside a single Super Logic Region, such as for $P \leq 16$ on U280. The rest of the data points used the default Vivado directives without any aggressive optimisation, with the exception of 512-bit switch designs for $P = 8/S = 2$ and $P = 16/S = 2$ and $S = 6$, which were initially outliers, possibly relating to the performance of the tool heuristics in place-and-route.

In the following section, we select our 16×16 switch implementations for comparison to the related work.

7.4 Comparison to FPGA-Based Related Work

In this section, we compare with the two FPGA-based switches mentioned in the related work (Section 3). The implementations here were redeveloped for the same target FPGA (Xilinx Alveo U280). Note that this comparison is mostly indicative, as some assumptions were made for each implementation, such as to be out-of-context as well (no actual transceiver is populated) and featuring different buffer lengths and memory types. There could also be additional effort to optimise the implementation of the competing designs.

Figure 24 shows how the design space for the proposed design performs against the related work, based on the throughput and port-to-port latency. For a packet size of 256 bits and $S = 4$, our proposal achieves a throughput of 53 Gbps, which is around 8.5 and 4.6 times the throughput of the 256-bit versions of SMiSLIP and GCQ, respectively. SMiSLIP seems to sometimes marginally win for port-to-port latencies, but our design space features choices that are better in both the throughput and latency, including the aforementioned data point, which has a latency of 24.2 ns.

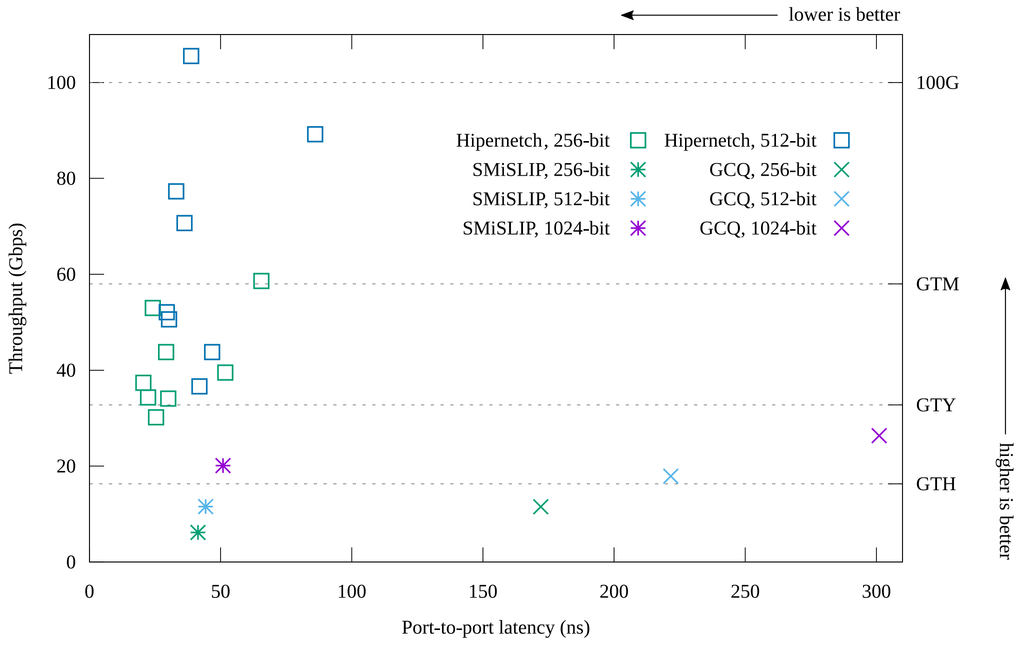


Fig. 24. Comparison of 16 × 16 switch implementations on Xilinx Alveo U280.

Table 2. Resource Utilisation of Related Work for P = 16 on Xilinx Alveo U280

	GCQ [18]		SMiSLIP [42]	
Packet Size (bits)	LUTs	FFs	LUTs	FFs
256	60,044	13,518	80,283	23,133
512	98,179	13,865	123,846	24,534
1024	222,964	85,982	212,778	25,539

In addition, since SMiSLIP uses the iSLIP scheduling algorithm, there will be additional latency for high input rates, as found in our simulations. GCQ yields the worst latencies irrespective of packet size. For a packet size of 512 bits and S = 2, we achieve the highest throughput overall at 105.5 Gbps and a latency of 39 ns, saturating the UltraScale+ GTY and GTM transceiver, as well as for 100 Gbps or equivalent.

In terms of hardware resources utilisation, Table 2 presents the Vivado-reported results of the implementation of the selected related work on the Alveo U280 FPGA board. At a first glance, it is apparent that in comparison with our approach, ours can generally require more resources. For example, our 256-bit, S = 4 implementation requires 173K LUTs and 155K FFs. However, if we take into account the port throughput of this data point (53 Gbps), we identify that the most appropriate competitors are the implementations with 1,024-bit packets. The 1,024-bit GCQ requires 1.3× and 0.55× the LUTs and FFs, respectively, whereas the SMiSLIP equivalent uses 1.2× the LUTs but less than 1/5 of the FFs. Note that even when using wider packets, the competing approaches are shown to be inferior in terms of throughput, latency and algorithmic performance. According to the system requirements, given the improved the performance of our approach overall, it can be worth the increased register utilisation from the pipelined approach.

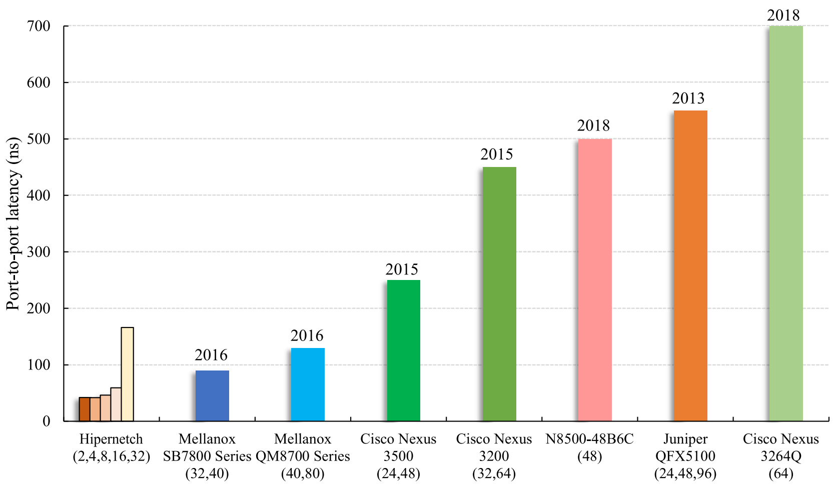


Fig. 25. Latency comparison with commodity switches [14–16, 35, 40, 41, 47] (from left to right).

7.5 Comparison to Commodity Switches

To compare the latency of our design with commodity switches, we collect latency values reported in the datasheet from well-known switch manufacturers, and the comparison is presented in Figure 25. Note that for commodity switches, most vendors provide the best latency among a series of products, and the relationship between the best latency and number of ports as well as the line speed is unknown. Thus, our reported best latency is chosen from designs of different line speed, given a fixed number of ports. The numbers in parentheses are the available number of ports supported by the corresponding switch.

As we focus on a single FPGA design, the number of ports is rather limited, and therefore we choose 1U commodity switches for a fairer comparison. Our design is shown to yield very low latencies, but it has the most basic port forwarding function and is implemented in our switch with custom packet format. As our design only considers the latency inside the FPGA, it is not appropriate to compare with commodity switches without the same interface logic (SFP+/QSFP+). Thus, we add the corresponding latency of such interface to our design. Taking the Ethernet switch as an example, we make an estimation using the 10G/25G Ethernet Subsystem from Xilinx [57], which gives the lowest latency of 36.8 ns and 46.08 ns for 10GE and 25GE design, respectively. To achieve the lowest latency, we choose 36.8 ns for our estimation.

To compare with other Ethernet switches, the port-to-port latency was updated as explained and corresponds to a very basic implementation with no additional functionality. As can be seen in the figure, our design has the lowest latency among the ASIC-based commodity switches, but for up to 16 ports. For the 32 ports, there is a noticeable performance overhead from approaching the limits of the FPGA device related to the Stacked Silicon Interconnect (SSI).

Due to the long iteration time of ASIC products, commodity switches tend to pack functions that might not be used by the customer in the switch. These additional functions potentially affect the latency of the switch. In contrast, for an FPGA-based switch design, it is the choice of the user to add to their desired functions. Therefore, FPGAs can minimise the latency overhead introduced by the redundant functions. Observing the latency difference between our design and the slowest commodity switch, our switch provides a time window of approximately 600 ns for the user to include additional functions like packet processing, and still able to compete with the commodity switches. Moreover, there will potentially be a frequency boost when transitioning from FPGA to ASIC implementations [30].

One limitation of FPGA-based switches is the number of ports that a single FPGA can support. Although our design shows the ability to saturate the fastest transceiver bandwidth on FPGAs, it can be challenging to utilise all of the available transceivers on board, due to routing and other resource limitations. However, it is possible to overcome such limitations by utilising multiple switches, or in different FPGA, hierarchically. Network topologies such as Clos networks [17] and fat-trees [32] are typical methods to construct large switching networks with smaller switching modules. Such methods can increase the scalability of FPGA-based switch designs at the expense of latency. These can also be regarded as a workaround for bigger switches before improved generations of FPGA devices are released.

8 FUTURE WORK

As future work, it would be useful to explore other functionalities of network switches on Hipernetch or a similar design. 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 is implemented out-of-context, but it could be applied on a specialised board with multiple high-speed transceivers as well as memories. Additionally, it is possible to explore the benefits of high bandwidth memory on the U280 board. With the built-in 32×32 AXI switch network in the high bandwidth memory controller [58], a larger switch with more functionality could be built on a single FPGA.

The proposed design is not necessarily restricted to implementation on FPGAs, and all proposed optimisations can also be applied in ASIC or ASSP implementations. Therefore, given its complexity and the benefits of reconfigurable logic, FPGAs can still remain in an end product, but maybe only for implementing additional functionality.

A category of possible future relates to the flexibility of FPGAs. By implementing the network switch on the FPGA, or introducing reconfigurable logic near a hardened network switch, a variety of interesting applications could be accelerated. By introducing computation units to the link layer, the data movement in distributed algorithms can be reduced. Example applications include data pre-processing [43], aggregation [56] for distributed machine learning, and stream processing [49] for big data analytics. Additionally, computational tasks that perform matrix multiplication, such as in encryption [25, 61] and compression [20, 31, 54], can benefit from the high speed DSPs in FPGAs. Customised protocols [11] that facilitate customised accelerators can also be added to an FPGA-based switch.

Due to the sequential transmission nature of the packets in our design, online computation can be supported by extending the pipeline. If the network switch is implemented on an FPGA, then partial-reconfiguration could be appropriate for swapping "plug-ins". Otherwise, small reconfigurable regions could be added to a hardened switch approach, with a resemblance to emerging techniques on adding reconfigurable instructions to CPUs [19, 29, 50]. Such "plug-ins" would introduce a fixed computation latency being added to the transmission latency, and help accelerating novel distributed computation tasks in the datacenter.

Today's switches are systems purposely built for the task and can provide links with a bandwidth of 400 Gbps [7], much higher than the the 58 Gbps supported on the latest FPGA device [59], although ports can also be combined [55]. 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 algorithmic solutions can contribute to a higher average port-to-port latency in all evaluated traffic patterns.

There could also be a wider design space exploration based on our register removal optimisation. For example, an unexplored optimisation is to use a different values of S for different pipeline stages groups. In this way, the optimisation could focus on pipeline stages with more hardware complexity. In Figure 6, where the building blocks are visualised, we can observe that there are varying levels of wire density. The complex pipeline stages could include the barrel shifters which seem more dense.

Another related 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 one 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.

Outside of networks, there could be many use cases where backpressure is needed. That is to enable a *ready* signal functionality, to avoid dropped packets. This could be useful for use in places where dropped packets are generally not allowed, such as for on-chip inter-core communication and interconnects. For networks, dropped packets were not of concern, due to the nature of the TCP and UDP protocols that are destined for large-scale deployments, and such events are assumed. A backpressure mechanism can be implemented, for example, by stalling the entire pipeline when there is no empty queue for the 'current' packets to be saved. In this case, it seems that the register reduction optimisation could also help to simplify the design, as less stall signals would be broadcasted overall.

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

9 CONCLUSION

This article presents Hipernetch, a novel high-performance network switch architecture for FPGAs. The main element is the *combined parallel round arbiter*, a fully pipelineable structure with which packets are rearranged at line rate, capable of filling the output queues in a 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 an Alveo U280 card, achieving an aggregate bandwidth of 1.7 Tbps, with more than 100 Gbits per port, while providing competitive switching performance for a wide range of traffic patterns.

ACKNOWLEDGMENTS

We thank Xilinx for the donation of the Alveo U280 board that was used in our evaluation. The authors would like to thank Behrad Niazmand for his insightful feedback on an early draft, as well as Chris Brooks and Rosie Prior from dunnhumby for their involvement in the partnership program. We also thank the anonymous reviewers, whose feedback influenced the future work section considerably.

REFERENCES

[1] OMNeT++. 2020. OMNeT++ Discrete Event Simulator. Retrieved September 21, 2021 from https://omnetpp.org/.

[2] 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, New York, NY, 98–107.

[3] Hamid Ahmadi, Wolfgang E. Denzel, Charles A. Murphy, and Erich Port. 1989. A high-performance switch fabric for integrated circuit and packet switching. *International Journal of Digital & Analog Cabled Systems* 2, 4 (1989), 277–287.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *SIGCOMM Computer Communication Review* 38, 4 (Aug. 2008), 63–74. https://doi.org/10.1145/1402946.1402967

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'13)*.

[6] 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* 11, 4 (1993), 319–352.

[7] Arista Networks, Inc. 2019. *7800R3 Series Data Center Switch Router Data Sheet*. Arista Networks, Inc.

[8] Kenneth E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference*. ACM, New York, NY, 307–314.

[9] 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 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, New York, NY, 89–100.

[10] Daniel Bundala and Jakub Závodnỳ. 2014. Optimal sorting networks. In *Proceedings of the International Conference on Language and Automata Theory and Applications*. 236–247.

[11] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, Los Alamitos, CA, 1–13.

[12] Jonathan Chao. 2000. Saturn: A terabit packet switch using dual round robin. *IEEE Communications Magazine* 38, 12 (2000), 78–84.

[13] 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.

[14] Cisco. 2020. Compare Models. Retrieved September 21, 2021 from https://www.cisco.com/c/en/us/products/switches/nexus-3000-series-switches/models-comparison.html#~tab-nexus3200.

[15] Cisco. 2020. Cisco Nexus 3000 Series Switches: Compare Models. Retrieved September 21, 2021 from https://www.cisco.com/c/en/us/products/switches/nexus-3000-series-switches/models-comparison.html#~tab-nexus3500.

[16] Cisco. 2020. Cisco Nexus 3264Q Switch Data Sheet. Retrieved September 21, 2021 from https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3264q-switch/datasheet-c78-734905.html.

[17] Charles Clos. 1953. A study of non-blocking switching networks. *Bell System Technical Journal* 32, 2 (1953), 406–424.

[18] 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, New York, NY, 67–76.

[19] Nguyen Dao, Andrew Attwood, Bea Healy, and Dirk Koch. 2020. FlexBex: A RISC-V with a reconfigurable instruction extension. In *Proceedings of the 2020 International Conference on Field-Programmable Technology (ICFPT'20)*.

[20] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA, 52–59.

[21] Nadeen Gebara, Jiuxi Meng, Wayne Luk, and Paolo Costa. 2018. Scheduling algorithms for high performance network switching on FPGAs: A survey. In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT'18)*. IEEE, Los Alamitos, CA, 166–173.

[22] Pankaj Gupta and Nick McKeown. 1999. Designing and implementing a fast crossbar scheduler. *IEEE Micro* 19, 1 (1999), 20–28.

[23] Chunzhi He and Kwan L. Yeung. 2011. D-LQF: An efficient distributed scheduling algorithm for input-queued switches. In *Proceedings of the 2011 IEEE International Conference on Communications (ICC'11)*. IEEE, Los Alamitos, CA, 1–5.

[24] W. Daniel Hillis and Guy L. Steele Jr. 1986. Data parallel algorithms. *Communications of the ACM* 29, 12 (1986), 1170–1183.

[25] Alireza Hodjat and Ingrid Verbauwhede. 2004. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA, 308–309.

[26] C. Hopps. 2000. Analysis of an Equal-Cost Multi-Path Algorithm. Retrieved September 21, 2021 from https://tools.ietf.org/html/rfc2992#: :text=Abstract%20Equal%2Dcost%20multi%2Dpath,method%20for%20making%20that%20decision.

[27] Yossi Kanizo, David Hay, and Isaac Keslassy. 2009. The crosspoint-queued switch. In *Proceedings of IEEE INFOCOM 2009*. IEEE, Los Alamitos, CA, 729–737.

[28] John Kim, William J. Dally, Brian Towles, and Amit K. Gupta. 2005. Microarchitecture of a high-radix router. *ACM SIGARCH Computer Architecture News* 33 (2005), 420–431.

[29] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. 2021. FABulous: An embedded FPGA framework. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 45–56.

[30] Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 26, 2 (2007), 203–215.

[31] Enas Dhuhri Kusuma and Thomas Sri Widodo. 2010. FPGA implementation of pipelined 2D-DCT and quantization architecture for JPEG image compression. In *Proceedings of the 2010 International Symposium on Information Technology*, Vol. 1. IEEE, Los Alamitos, CA, 1–6.

[32] Charles E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers* 100, 10 (1985), 892–901.

[33] Yihan Li, Shivendra Panwar, and H. Jonathan Chao. 2001. On the performance of a dual round-robin switch. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society (INFOCOM'01)*, Vol. 3. IEEE, Los Alamitos, CA, 1688–1697.

[34] Yihan Li, Shivendra Panwar, and H. Jonathan Chao. 2002. The dual round robin matching switch with exhaustive service. In *Proceedings of the Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*. IEEE, Los Alamitos, CA, 58–63.

[35] FIBERSTORE Ltd. 2020. N8500-48B6C 48-Port 25Gb SFP28 L3 Trident 3 Data Centre Managed Ethernet Switch. Retrieved September 21, 2021 from https://www.fs.com/uk/products/75807.html.

[36] Robert B. Magill, Charles E. Rohrs, and Robert L. Stevenson. 2003. Output-queued switch emulation by fabrics with limited memory. *IEEE Journal on Selected Areas in Communications* 21, 4 (2003), 606–615.

[37] J. Mahdavi, S. Floyd, and A. Romanow. 1996. TCP Selective Acknowledgment Options. Retrieved September 21, 2021 from https://tools.ietf.org/html/rfc2992#:~:text=Abstract%20Equal%2Dcost%20multi%2Dpath,method%20for%20making%20that%20decision.

[38] Nick McKeown. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking* 2 (1999), 188–201.

[39] Nick McKeown, Balaji Prabhakar, and Mingyan Zhu. 1997. Matching output queueing with combined input and output queueing. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, Vol. 35. 595–603.

[40] Mellanox. 2020. QM8700 Mellanox Quantum™ HDR Edge Switch. Retrieved September 21, 2021 from https://www.mellanox.com/sites/default/files/doc-2020/pb-qm8700.pdf.

[41] Mellanox. 2020. SB7800 InfiniBand EDR 100Gb/s Switch System. Retrieved September 21, 2021 from https://www.mellanox.com/sites/default/files/doc-2020/pb-sb7800.pdf.

[42] Jiuxi Meng, Nadeen Gebara, Ho-Cheung Ng, Paolo Costa, and Wayne Luk. 2019. Investigating the feasibility of FPGA-based network switches. In *Proceedings of the 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'19)*. IEEE, Los Alamitos, CA.

[43] Jiuxi Meng, Ce Guo, Nadeen Gebara, and Wayne Luk. 2020. Fast and accurate training of ensemble models with FPGA-based switch. In *Proceedings of the 2020 IEEE 31st International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'20)*. IEEE, Los Alamitos, CA, 81–84.

[44] Lotfi Mhamdi and Mounir Hamdi. 2003. Output queued switch emulation by a one-cell-internally buffered crossbar switch. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'03)*, Vol. 7. IEEE, Los Alamitos, CA, 3688–3693.

[45] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting networks on FPGAs. *VLDB Journal—The International Journal on Very Large Data Bases* 21, 1 (2012), 1–23.

[46] 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.

[47] Juniper Networks. 2019. QFX5100 Ethernet Switch. Retrieved September 21, 2021 from https://www.juniper.net/assets/us/en/local/pdf/datasheets/1000480-en.pdf.

[48] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2018. FLiMS: Fast lightweight merge sorter. In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT'18)*. IEEE, Los Alamitos, CA, 78–85.

[49] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2020. An adaptable high-throughput FPGA merge sorter for accelerating database analytics. In *Proceedings of the 2020 30th International Conference on Field Programmable Logic and Applications (FPL'20)*. IEEE, Los Alamitos, CA, 65–72.
[50] Philippos Papaphilippou, Paul H. J. Kelly, and Wayne Luk. 2021. Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions. In *Proceedings of the 5th Workshop on Computer Architecture Research with RISC-V (CARRV'21), Held in Conjuction with ISCA 2021.*
[51] Philippos Papaphilippou and Wayne Luk. 2018. Accelerating database systems using FPGAs: A survey. In *Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. IEEE, Los Alamitos, CA, 125–130.
[52] Philippos Papaphilippou, Jiuxi Meng, and 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)*. ACM, New York, NY, 76–85. https://doi.org/10.1145/3373087.3375299
[53] Philippos Papaphilippou, Holger Pirk, and Wayne Luk. 2019. Accelerating the merge phase of sort-merge join. In *Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, Los Alamitos, CA, 100–105.
[54] Suzanne Rigler, William Bishop, and Andrew Kennings. 2007. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *Proceedings of the 2007 Canadian Conference on Electrical and Computer Engineering*. IEEE, Los Alamitos, CA, 1235–1238.
[55] 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 *Proceedings of the 2019 30th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, Los Alamitos, CA.
[56] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2020. Scaling distributed machine learning with in-network aggregation. arXiv:cs.DC/1903.06701
[57] Xilinx. 2019. 10G/25G High Speed Ethernet Subsystem v3.0. Retrieved September 21, 2021 from https://www.xilinx.com/support/documentation/ip_documentation/xxv_ethernet/v3_0/pg210-25g-ethernet.pdf.
[58] Xilinx. 2019. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance. Retrieved September 21, 2021 from https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf.
[59] Xilinx Inc. 2015–2019. *UltraScale+ FPGA Product Tables and Product Selection Guide*. Xilinx Inc.
[60] Xilinx Inc. 2019. *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics (DS923)*. Xilinx Inc.
[61] Yulin Zhang and Xinggang Wang. 2010. Pipelined implementation of AES encryption based on FPGA. In *Proceedings of the 2010 IEEE International Conference on Information Theory and Information Security*. IEEE, Los Alamitos, CA, 170–173.
[62] Si Qing Zheng, Mei Yang, John Blanton, Prasad Golla, and Dominique Verchere. 2002. A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling. In *Proceedings of the 2002 45th Midwest Symposium on Circuits and Systems (MWSCAS'02)*, Vol. 2. IEEE, Los Alamitos, CA, II.
[63] Moshe Zukerman. 2013. Introduction to queueing theory and stochastic teletraffic models. arXiv:1307.2968

Received July 2020; revised March 2021; accepted July 2021