Design and Applications for Embedded Networks-on-Chip on FPGAs

Mohamed S. Abdelfattah, *Member, IEEE,* Andrew Bitar, *Member, IEEE,* Vaughn Betz *Member, IEEE*

Abstract— Field-programmable gate-arrays (FPGAs) have evolved to include embedded memory, high-speed I/O interfaces and processors, making them both more efficient and easier-to-use for compute acceleration and networking applications. However, implementing on-chip communication is still a designer's burden wherein custom system-level buses are implemented using the fine-grained FPGA logic and interconnect fabric. Instead, we propose augmenting FPGAs with an embedded network-on-chip (NoC) to implement system-level communication. We design custom interfaces to connect a packet-switched NoC to the FPGA fabric and I/Os in a configurable and efficient way and then define the necessary conditions to implement common FPGA design styles with an embedded NoC. Four application case studies highlight the advantages of using an embedded NoC. We show that access latency to external memory can be ~1.5× lower. Our application case study with image compression shows that an embedded NoC improves frequency by 10–80%, reduces utilization of scarce long wires by 40% and makes design easier and more predictable. Additionally, we leverage the embedded NoC in creating a programmable Ethernet switch that can support up to 819 Gb/s – 5× more switching bandwidth and 3× lower area compared to previous work. Finally, we design a 400 Gb/s NoC-based packet processor that is very flexible and more efficient than other FPGA-based packet processors.

Index Terms—Field-programmable gate-array, network-on-chip, latency-insensitive, image compression, networking, packet processing.

1 Introduction

FIELD-programmable gate-arrays (FPGAs) are an effective compute acceleration platform for datacenter [1] and networking [2] applications. Besides the configurable FPGA fabric, modern FPGAs contain floating-point arithmetic units, embedded processor cores, and hard controllers for external memory, PCIe and Ethernet [3]. These embedded resources greatly enhance FPGA computation and data transfer through I/Os; however, on-chip communication has been little developed in the past two decades. The traditional FPGA interconnect consists of wire segments and switches. While this is very flexible in creating custom fine-grained connections, it is inefficient for constructing wide buses for high-bandwidth data transfer across the chip.

Fig. 1 illustrates a sample FPGA application that is connected using a soft multiplexed bus, created using the FPGA's traditional interconnect and logic fabric. To create wide connections that are typically hundreds of bits wide, each bit is stitched together from the FPGA's wire segments and configurable interconnect switches. Additionally, the soft FPGA logic fabric is used to create multiplexers or crossbars to switch data between multiple application modules. These buses are difficult to design for many reasons. The physical size of the bus is only known after a design is completed; consequently, its area and power consumption – which are typically large [4] – and speed are unpredictable until the very last stages of compilation. As FPGAs scale to larger capacities, and wire speed deteriorates [5], it is more challenging to design a bus that meets the speed requirements of FPGA applications, especially the stringent timing requirements of high speed I/O interfaces like external memory. If the soft bus does not meet the application's speed targets, time-consuming design patches are required to add pipeline registers or rethink the bus architecture to make it fast enough. Since FPGA applications take hours or days for synthesis, placement and routing, these timing closure iterations are very inefficient and greatly hamper design productivity with FPGAs [6]. New FPGAs now contain pipeline registers in their programmable interconnect which makes timing closure easier [7]; however, designers must still redesign their system-level interconnect to suit each new application or FPGA device.

These limitations of soft buses are a barrier to FPGA adoption in mainstream computing; therefore, our goal is to abstract system-level communication using an embedded network-on-chip (NoC) as shown in Fig. 1. By prefabricating the NoC, its speed is known before design compilation thus mitigating or eliminating timing closure iterations. Additionally, the embedded NoC uses less area and power compared to soft buses for most FPGA applications [4], [6]. An embedded NoC improves design portability across different applications or devices since the system-level interconnect becomes built into the FPGA, and the application designer needs only to focus on creating the application kernels. Importantly, an NoC decouples the application's computation and communication logic. This improves design modularity, relaxes placement and routing constraints, and enables the independent optimization of application modules, which not only simplifies design but can also improve performance. Improved design modularity can also lead to easier parallel compilation and partial reconfiguration flows.

To reap the potential benefits of embedded NoCs without losing configurability – the hallmark of FPGAs – we propose flexible interfaces between the embedded NoC and the FPGA

• *The authors are with the Department of Electrical and Computer Engineering, University of Toronto.*
E-mails: {mohamed,bitar,vaughn}@eecg.utoronto.ca

Manuscript received April 30, 2016; revised August 31, 2016.

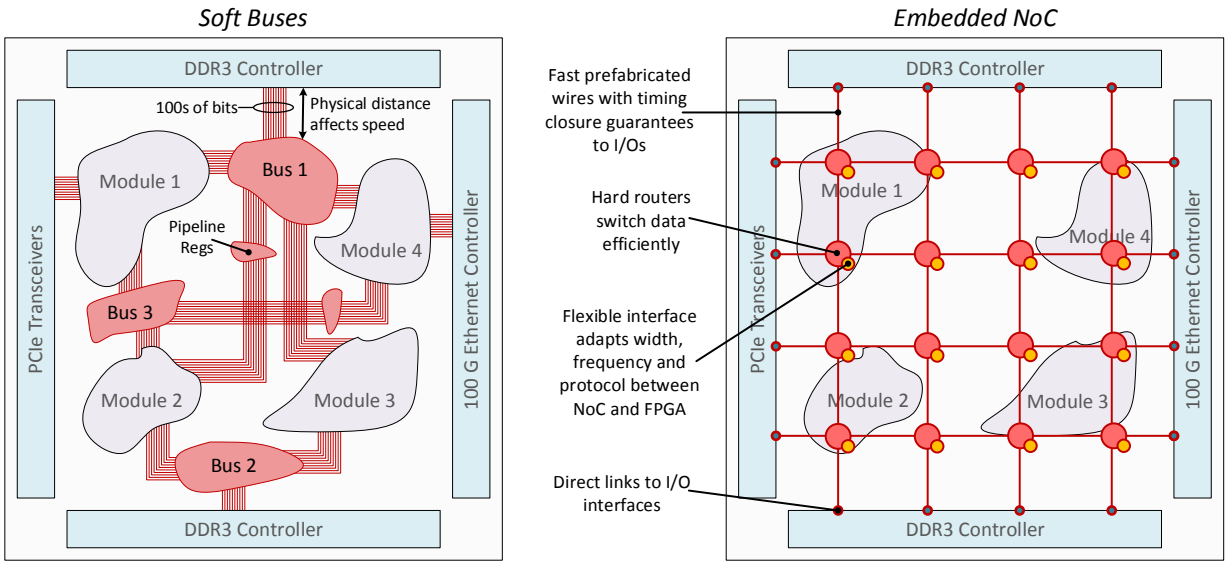


Fig. 1: System-level interconnection on FPGAs with soft buses or an embedded NoC.

fabric and I/Os. Furthermore, we define rules that make FPGA design styles compatible with the embedded NoC. We also present four application case studies to highlight the utility of an embedded NoC in important and diverse FPGA applications. To this end, we make the following contributions:

1) Present the FabricPort: a flexible interface between the FPGA fabric and a packet-switched embedded NoC.
2) Present IOLinks: direct connections between the embedded NoC and the FPGA's memory and I/O controllers.
3) Enumerate the conditions for semantically correct FPGA communication using the embedded NoC.
4) Present `RTL2Booksim`: allowing the co-simulation of a software NoC simulator and hardware RTL designs.
5) Compare the latency of external memory access with an embedded NoC (with IOLink) or a soft bus.
6) Analyze latency-sensitive parallel JPEG compression both with and without an embedded NoC.
7) Design an Ethernet switch with 5× more bandwidth at 3× less area compared to previous FPGA switches.
8) Design a more flexible and efficient FPGA packet processor using the embedded NoC.

2 Embedded Hard NoC

Our embedded packet-switched NoC targets a large 28 nm FPGA device. The NoC presented in this section is used throughout this paper in our design and evaluation sections. We base our router design on a state-of-the-art packet-switched router that uses credit-based flow control and virtual channels (VCs) [8].

In designing the embedded NoC, we must over-provision its resources, much like other FPGA interconnect resources, so that it can be used in connecting *any* application. We therefore look at high bandwidth I/O interfaces to determine the required NoC link bandwidth. The highest-bandwidth interface on FPGAs is usually a DDR3 interface, capable of transporting 64 bits of data at a speed of 1067 MHz at double-data rate (~17 GB/s). We design the NoC such that it can transport the entire bandwidth of a DDR3 interface on one of its links; therefore, we can connect to DDR3 using a single router port. Additionally, we must be able to transport the control data of DDR3 transfers, such as the address, alongside the data. We therefore choose a width of 150 bits for our NoC links and router ports, and we are able to run the NoC at 1.2 GHz[1] [10]. By multiplying our width and frequency, we find that our NoC is able to transport a bandwidth of 22.5 GB/s on each of its links.

TABLE 1: NoC parameters and properties for 28 nm FPGAs.

NoC Link Width	# VCs	Buffer Depth	# Nodes	Topology
150 bits	2	10 flits/VC	16 nodes	Mesh

Area†	Area Fraction	Frequency
528 LABs	1.3%	1.2 GHz

†LAB: Area equivalent to a Stratix V logic cluster.

Table 1 summarizes the NoC parameters and properties. We leverage VCs to avoid deadlock, and merge data streams as we discuss in Section 4. Additionally, we believe that the capabilities offered by VCs – such as assigning priorities to different messages types – would be useful in future FPGA designs. The buffer depth per VC is provisioned such that it is not a cause for throughput degradation by sizing our router buffers to cover the *credit round-trip latency* [11]. With the given parameters, each embedded router occupies an area equivalent to 35 logic clusters (Stratix-V LABs), including the interface between the router and the FPGA fabric, and including the wire drivers necessary for the hard NoC links [12]. As Table 1 shows, the whole 16-node NoC occupies 528 LABs, a mere 1.3% of a large 28 nm Stratix-V FPGA core area (excluding I/Os).

1. We implement the NoC in 65 nm standard cells and scale the frequency obtained by 1.35× to match the speed scaling of Xilinx's (also standard cell) DSP blocks from Virtex5 (65 nm) to Virtex7 (28 nm) [9].

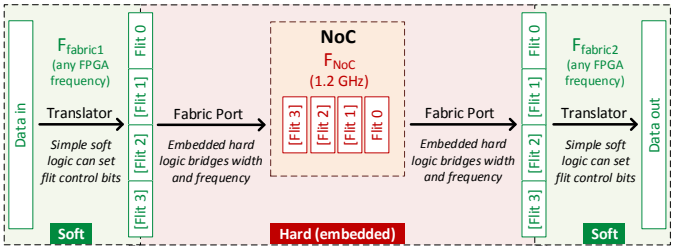


Fig. 2: Data on the FPGA with any protocol can be translated into NoC flits using application-dependent soft logic (translator). A FabricPort adapts width (1-4 flits on fabric side and 1 flit on NoC) and frequency (any frequency on fabric side and 1.2 GHz on NoC side) to inject flits into the NoC.

3 FPGA-NoC Interfaces

This section describes the circuitry between our embedded NoC and the FPGA's fabric and I/Os. The FabricPort is a flexible interface that connects to the FPGA fabric, and IOLinks are direct connections to I/O interfaces.

3.1 FabricPort

Each NoC port can sustain a maximum input bandwidth of 22.5 GB/s; however, this is done at the high frequency of 1.2 GHz for our NoC. The main purpose of the FabricPort is therefore to give the FPGA fabric access to that communication bandwidth, at the range of frequencies at which FPGAs normally operate. How does one connect a module configured from the FPGA fabric to the embedded NoC running at a different width and frequency?

Fig. 2 illustrates the process of conditioning data from any FPGA module to NoC flits, and vice versa. A very simple translator takes incoming data and appends to it the necessary flit control information. For most cases, this translator consists only of wires that pack the data in the correct position and sets the valid/head/tail bits from constants. Once data is formatted into flits, we can send between 0 and 4 flits in each fabric cycle, this is indicated by the valid bit on each flit. The FabricPort will then serialize the flits, one after the other, and inject the valid ones into the NoC at the NoC's frequency. When flits are received at the other end of the NoC, the frequency is again bridged, and the width adapted using a FabricPort; then a translator strips control bits and injects the data into the receiving module.

This FabricPort plays a pivotal role in adapting an embedded NoC to function on an FPGA. We must bridge the width and frequency while making sure that the FabricPort is never a source of throughput reduction; furthermore, the FabricPort must be able to interface to different VCs on the NoC, send/receive different-length packets and respond to backpressure coming from either the NoC or FPGA fabric. We enumerate the essential properties that this component must have:

1) **Rate Conversion**: Match the NoC bandwidth to the fabric bandwidth. Because the NoC is embedded, it can run ~4× faster than the FPGA fabric [12]. We leverage that speed advantage to build a narrow-link-width NoC that connects to a wider but slower FPGA fabric.
2) **Stallability**: Accept/send data on every NoC cycle in the absence of stalls, and stall for the minimum number of cycles when the fabric/NoC isn't ready to send/receive data. The FabricPort itself should never be the source of throughput reduction.
3) **Virtual Channels**: Read/write data from/to multiple virtual channels in the NoC such that the FabricPort is never the cause for deadlock.
4) **Packet Length**: Transmit packets of different lengths.
5) **Backpressure Translation**: Convert credit-based flow-control into the more FPGA-familiar ready/valid.

3.1.1 FabricPort Input: Fabric→NoC

Fig. 3 shows a schematic of the FabricPort with important control signals annotated. The FabricPort input (Fig. 3a) connects the output of a module in the FPGA fabric to an embedded NoC input. Following the diagram from left to right: data is input to the time-domain multiplexing (TDM) circuitry on each fabric clock cycle and is buffered in the "main" register. The "aux" register is added to provide elasticity. Whenever the output of the TDM must stall there is a clock cycle before the stall signal is processed by the fabric module. In that cycle, the incoming datum may still be valid, and is therefore buffered in the "aux" registers. Importantly, this stall protocol ensures that every stall (ready = 0) cycle only stops the input for exactly one cycle ensuring that the FabricPort input does not reduce throughput.

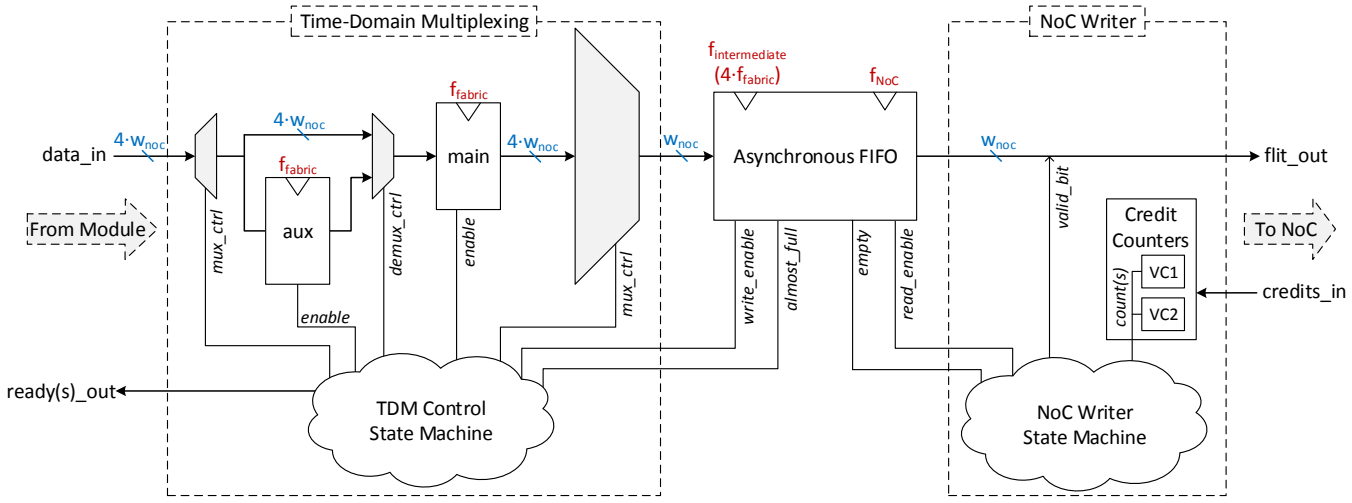
The TDM unit takes four flits input on a slow fabric clock and outputs one flit at a time on a faster clock that is 4× as fast as the FPGA fabric – we call this the intermediate clock. This intermediate clock is only used in the FabricPort between the TDM unit and the asynchronous FIFO (aFIFO) buffer. Because it is used only in this very localized region, this clock may be derived locally from the fabric clock by careful design of circuitry that multiplies the frequency of the clock by four. This is better than generating 16 different clocks globally through phase-locked loops, then building a different clock tree for each router's intermediate clock (a viable but more costly alternative).

The output of the TDM unit is a new flit on each intermediate clock cycle. Because each flit has a valid bit, only those flits that are valid will actually be written in the aFIFO thus ensuring that no invalid data propagates downstream, unnecessarily consuming power and bandwidth. The aFIFO bridges the frequency between the intermediate clock and the NoC clock ensuring that the fabric clock can be completely independent from the NoC clock frequency and phase.

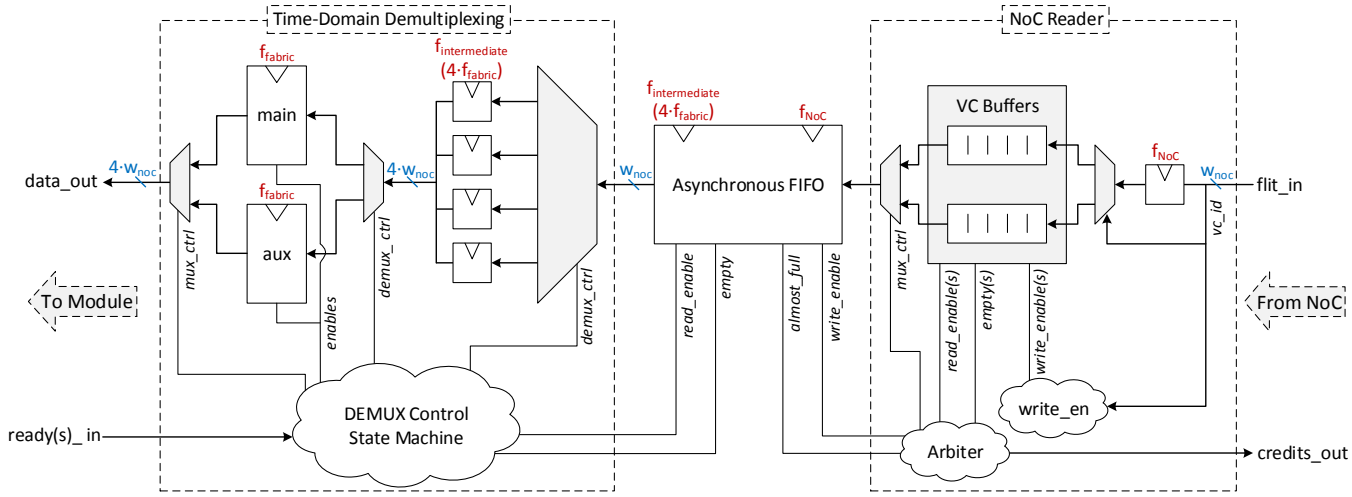
The final component in the FabricPort input is the "NoC Writer". This unit reads flits from the aFIFO and writes them to the downstream NoC router. The NoC Writer keeps track of the number of credits in the downstream router to interface to the credit-based backpressure system in the embedded NoC, and only sends flits when there are available credits. Note that credit-based flow control is by far the most-widely-used backpressure mechanism in NoCs because of its superior performance with limited buffering [11].

3.1.2 FabricPort Output: NoC→Fabric

Fig. 3b details a FabricPort output; the connection from an NoC output port to the input of a module on the FPGA fabric.



(a) FabricPort input: from the FPGA fabric to the embedded NoC.



(b) FabricPort output: from the embedded NoC to the FPGA fabric.

Fig. 3: The FabricPort interfaces the FPGA fabric to an embedded NoC in a flexible way by bridging the different frequencies and widths as well as handling backpressure from both the FPGA fabric and the NoC.

Following the diagram from right to left: the first component is the "NoC Reader". This unit is responsible for reading flits from an NoC router output port and writing to the aFIFO. Note that separate FIFO queues must be kept for each VC; this is very important as it avoids scrambling data from two packets. Fig. 4 clarifies this point; the upstream router may interleave flits from different packets if they are on different VCs. By maintaining separate queues in the NoC reader, we can rearrange flits such that flits of the same packet are organized one after the other.

The NoC reader is then responsible for arbitrating between the FIFO queues and forwarding one (entire) packet – one flit at a time – from each VC. We currently implement fair round-robin arbitration and make sure that there are no "dead" arbitration cycles. That means that as soon as the NoC reader sees a tail flit of one packet, it has already computed the VC from which it will read next. The packet then enters the aFIFO where it crosses clock domains between the NoC clock and the intermediate clock.

The final step in the FabricPort output is the time-domain demultiplexing (DEMUX). This unit reassembles packets (or packet fragments if a packet is longer than 4 flits) by combining 1-4 flits into the wide output port. In doing so, the DEMUX does not combine flits of different packets and will instead insert invalid zero flits to pad the end of a packet that doesn't have a number of flits divisible by 4 (see Fig. 4). This is very much necessary to present a simple interface for designers allowing them to connect design modules to the FabricPort with minimal soft logic.

3.2 IOLinks

The FabricPort interfaces between the NoC and the FPGA in a flexible way. To connect to I/O interfaces, such as external memory interfaces, we can connect through a regular Fabricport interface. This could be done by simply connecting the I/O interface to soft logic which then connects to a FabricPort as shown in Fig. 5a. However, the soft logic between an I/O interface and the FabricPort may be difficult to design for many reasons:

- Fast I/O interfaces have very stringent timing requirements, making timing closure on any soft logic connecting to it very challenging.

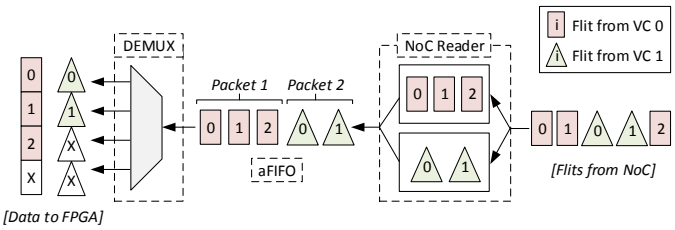
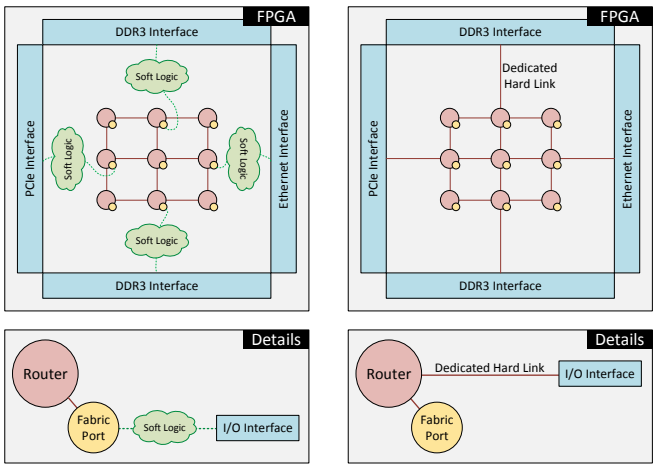


Fig. 4: "NoC Reader" sorts flits from each VC into a separate queue thereby ensuring that flits of each packet are contiguous. The DEMUX then packs up to four flits together and writes them to the wide output port but never mixes flits of two packets.



(a) Through the FabricPort. (b) Directly using hard IOLinks.

Fig. 5: Two options for connecting NoC to I/O interfaces.

- The NoC router may be physically placed far away from the I/O interface, thus heavily-pipelined soft logic is required to connect the two. This may incur significant area and power overhead as the data bandwidth of some I/O interfaces is very large, which translates into a wide data path in the slow FPGA logic. Furthermore, adding pipeline registers would improve timing but typically worsen latency – a critical parameter of transferring data over some I/Os.
- Any solution is specific to a certain FPGA device and will not be portable to another device architecture.

These shortcomings are the same ones that we use to motivate the use of an embedded NoC instead of a soft bus to distribute I/O data. Therefore, if we connect to I/O interfaces through the FabricPort, we lose some of the advantages of embedding a system-level NoC. Instead, we propose connecting NoC routers directly to I/O interfaces using hard links as shown in Fig. 5b. In addition, clock-crossing circuitry (such as an aFIFO) will be required to bridge the frequency of the I/O interface and the embedded NoC since they will very likely differ. We call these direct I/O links with clock crossing "IOLinks". They have many advantages:

- Uses fewer NoC resources since it frees a FabricPort which can be used for something else.
- Reduces soft logic utilization conserving area and power.
- Reduces data transfer latency between NoC and I/O interfaces because we avoid adding pipelined soft logic.

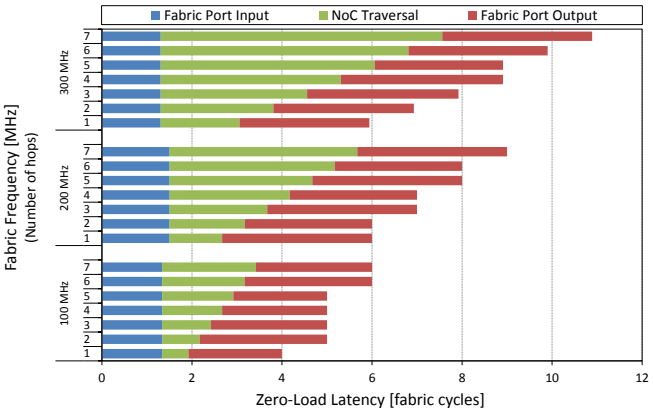


Fig. 6: Zero-load latency of the embedded NoC (including FabricPorts) at different fabric frequencies. Latency is reported as the number of cycles at each fabric frequency. The number of hops varies from 1 hop (minimum) to 7 hops (maximum – chip diagonal).

One possible shortcoming of IOLinks is the loss of reconfigurability – it is important to design these I/O links such that they do not rid the FPGA I/O interfaces of any of their built-in flexibility. IOLinks should be optional by using multiplexers in the I/O interfaces to choose between our IOLinks and the traditional interface to the FPGA logic. This will maintain the option of directly using I/O interfaces without using IOLinks, thus maintaining the configurability of I/O interfaces. Note that an IOLink's implementation will be very specific to the I/O interface to which it connects; we study an IOLink to DDR3 interface in Section 5.2.

4 Design Styles and Constraints with NoC

This section discusses conditions that are necessary to *adapt* an embedded NoC to function with FPGA design styles.

4.1 Latency and Throughput

Fig. 6 plots the zero-load latency of the NoC (running at 1.2 GHz) for different fabric frequencies that are typical of FPGAs. We measure latency by sending a single 4-flit packet through the FabricPort input→NoC→FabricPort output. The NoC itself is running at a very fast speed, so even if each NoC hop incurs 4 cycles of NoC clocks, this translates to approximately 1 fabric clock cycle. However, the FabricPort latency is a major portion of the total latency of data transfers on the NoC; it accounts for 40%–85% of latency in an unloaded embedded NoC. The reason for this latency is the flexibility offered by the FabricPort – we can connect a module of any operating frequency but that incurs TDM, DEMUX and clock-crossing latency.

Fig. 7 plots the throughput between any source and destination on our NoC in the absence of contention. The NoC is running at 1.2 GHz with 1-flit width; therefore, if we send 1 flit each cycle at a frequency lower than 1.2 GHz, our throughput is always perfect – we'll receive data at the same input rate (one flit per cycle) on the other end of the NoC path. The same is true for 2-flits (300 bits) at 600 MHz, 3 flits (450 bits) at 400 MHz or 4 flits (600 bits) at 300 MHz.

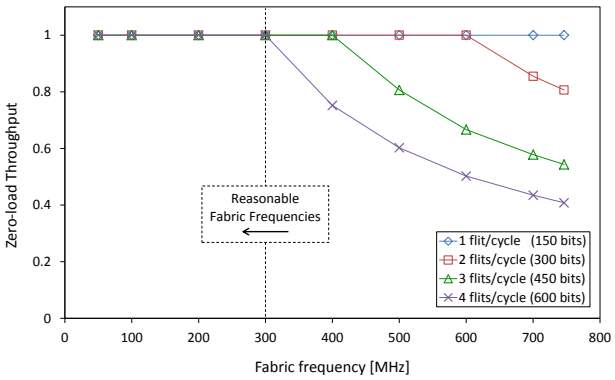


Fig. 7: Zero-load throughput of embedded NoC path between any two nodes, normalized to sent data. A throughput of "1" is the maximum; it means that we receive i flits per cycle, where i is the number of flits we send each cycle.

As Fig. 7 shows, the NoC can support the mentioned width–frequency combinations because they are different ways to utilize the NoC bandwidth. In 28-nm FPGAs, we believe that very few wide datapath designs will run faster than 300 MHz, therefore the NoC is very usable at all its different width options. When the width–frequency product exceeds the NoC bandwidth, packet transfers are still correct; however, the throughput degrades and the NoC backpressure stalls the data sender thus reducing throughput as shown in Fig. 7.

4.2 Module Connectivity

The FabricPort converts 22.5 GB/s of NoC link data bandwidth (150 bits, 1.2 GHz) to 600 bits and any fabric frequency on the fabric side. An FPGA designer can then use any fraction of that port width to send data across the NoC. However, the smallest NoC unit is the flit; so we can either send 1, 2, 3 or 4 flits each cycle. If the designer connects data that fits in one flit (150 bits or less), all the data transported by the NoC is useful data. However, if the designer want to send data that fits in one-and-a-half flits (225 bits for example), then the FabricPort will send two flits, and half of the second flit is overhead that adds to power consumption and worsens NoC congestion unnecessarily. Efficient "translator" modules (see Fig. 2) will therefore try to take the flit width into account when injecting data to the NoC.

A limitation of the FabricPort output is observed when connecting two modules. Even if each module only uses half the FabricPort's width (2 flits), only one module can receive data each cycle because the DEMUX only outputs one packet at a time by default as Fig. 4 shows. To overcome this limitation, we create a *combine-data* mode as shown in Fig. 8. For this combine-data mode, when there are two modules connected to one FabricPort, data for each module must arrive on a different VC. The NoC Reader arbiter must strictly alternate between VCs, and then the DEMUX will be able to group two packets (one from each VC) before data output to the FPGA. This allows merging two streams without incurring serialization latency in the FabricPort.

Condition 1. To combine packets at a FabricPort output, each packet must arrive on a different VC.

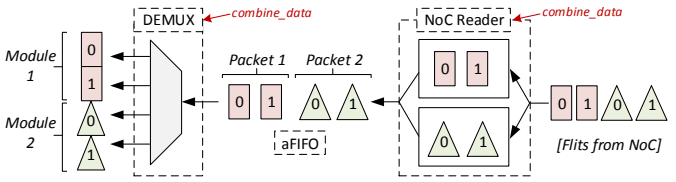


Fig. 8: FabricPort output merging two packets from separate VCs in *combine-data* mode, to be able to output data for two modules in the same clock cycle.

Note that we are limited to the merging of 2 packets with 2 VCs because each packet type must have the ability to be independently stalled, and we can only stall an entire VC, not individual flits within a single VC. We can merge up to four 1-flit packets if we increase the number of VCs.

4.3 Packet Ordering

Packet-switched NoCs like the one we are using were originally built for chip multiprocessors (CMPs). CMPs only perform **transaction** communication; most transfers are cache lines or coherency messages. Furthermore, processors have built-in mechanisms for reordering received data, and NoCs are typically allowed to reorder packets.

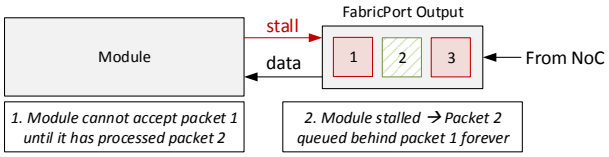
With FPGAs, transaction communication can be one of two main things: (1) Control data from a soft processor that is low-bandwidth and latency-critical – a poor target for embedded NoCs, or (2) Communication between design modules and on-chip or off-chip memory, or PCIe links – high bandwidth data suitable for our proposed NoC. Additionally, FPGAs are very good at implementing **streaming**, or data-flow applications such as packet switching, video processing, compression and encryption. These streams of data are also prime targets for using our high-bandwidth embedded NoC. Crucially, neither transaction nor streaming applications tolerate packet reordering on FPGAs, nor do FPGAs natively support it. While it may be possible to design reordering logic for simple memory-mapped applications, it becomes *impossible* to build such logic for streaming applications without hurting performance – we therefore choose to restrict the embedded NoC to perform in-order data transfers only. Specifically, an NoC is not allowed to reorder packets on a single connection.

Definition 1. A **connection** (**s**, **d**) exists between a single source (**s**) and its downstream destination (**d**) to which it sends data.

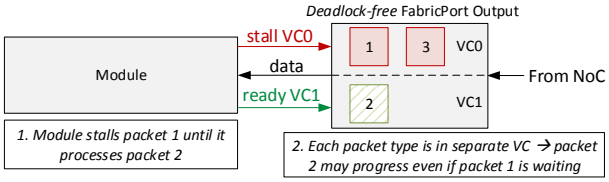
Definition 2. A **path** is the sequence of links from **s** to **d** that a flit takes in traversing an NoC.

There are two causes of packet reordering. Firstly, an adaptive route-selection algorithm would always attempt to choose a path of least contention through the NoC; therefore two packets of the same source and destination (same connection) may take different paths and arrive out of order. Secondly, when sending packets (on the same connection) but different VCs, two packets may get reordered even if they are both taking the same path through the NoC.

To solve the first problem, we only use deterministic routing algorithms, such as dimension-ordered routing [11],



(a) Standard FabricPort output.



(b) Deadlock-free FabricPort output.

Fig. 9: Deadlock can occur if a dependency exists between two packets going to the same port. By using separate VCs for each message type, deadlock can be broken thus allowing two dependent message types to share a FabricPort output.

in which paths are the same for all packets that belong to a connection.

Condition 2. The same **path** must be taken by all packets that belong to the same **connection**.

Eliminating VCs altogether would fix the second ordering problem; however, this is not necessary. VCs can be used to break message deadlock, merge data streams (Fig. 8), alleviate NoC congestion and may be also used to assign packet priorities thus adding extra configurability to our NoC – these properties are desirable. We therefore impose more specific constraints on VCs such that they may still be used on FPGA NoCs.

Condition 3. All packets belonging to the same **connection** that are of the same message type, must use the same VC.

4.4 Dependencies and Deadlock

Two *message types* may not share a standard FabricPort output (Fig. 3b) if a dependency exists between the two message types. An example of dependent message types can be seen in video processing IP cores: both control messages (that configure the IP to the correct resolution for example) and data messages (pixels of a video stream) are received on the same port [13]. An IP core may not be able to process the data messages correctly until it receives a control message.

Consider the deadlock scenario in Fig. 9a. The module is expecting to receive packet 2 but gets packet 1 instead; therefore it stalls the FabricPort output and packet 2 remains queued behind packet 1 forever. To avoid this deadlock, we can send each message type on a different VC [14]. Additionally, we created a deadlock-free FabricPort output that maintains separate paths for each VC beyond the NoC reader – this means we duplicate the aFIFO and DEMUX units for each VC we have. Each VC now has an independent "ready" signal, allowing us to stall each VC separately. Fig. 9b shows that even if there is a dependency between different messages, they can share a FabricPort output provided each uses a different VC.

Condition 4. When multiple message types can be sent to a FabricPort, and a dependency exists between the message types, each type must use a different VC. The number of dependencies must be less than or equal to the number of VCs.

4.5 Latency-Sensitive Design with NoC (Permapaths)

Communication over an NoC naturally has variable latency; however, latency-sensitive design requires predictable latency on the connections between modules. That means that the interconnect is not allowed to insert/remove any cycles between successive data. Prior NoC literature has largely focused on using circuit-switching to achieve quality-of-service guarantees but could only provide a bound on latency rather than a guarantee of fixed latency [15]. We analyze the latency and throughput guarantees that can be attained from an NoC, and use those guarantees to determine the conditions under which a latency-sensitive system can be mapped onto a packet-switched embedded NoC. Effectively, our methodology creates permanent paths with predictable latencies (Permapaths) on our embedded NoC.

A NoC connection acts as a pipelined wire; the number of pipeline stages are equivalent to the zero-load latency of an NoC path; however, that latency is only incurred once at the very beginning of data transmission after which data arrives on each fabric clock cycle. We call this a **Permapath** through the NoC: a path that is free of contention and has perfect throughput. As Fig. 7 shows, we can create Permapaths of larger widths provided that the input bandwidth of our connection does not exceed the NoC port bandwidth of 22.5 GB/s. This is why throughput is still perfect with 4 flits×300 MHz for instance. To create those Permapaths we must therefore ensure two things:

Condition 5. (Permapaths) The sending module data bandwidth must be less than or equal to the maximum FabricPort input bandwidth.

Condition 6. (Permapaths) No other data traffic may overlap the NoC links reserved for a Permapath to avoid congestion delays on those links.

Condition 6 can be determined statically since our routing algorithm is deterministic; therefore, the mapping of modules onto NoC routers is sufficient to identify which NoC links will be used by each module.

One final constraint is necessary to ensure error-free latency-sensitive transfers. It pertains to "clock drift" that occurs between the intermediate clock and the NoC clock – these are respectively the read and write clocks of the aFIFO in the FaricPort (Fig. 3). If these clocks are different, and they drift apart because of their independence, data may not be consistently latched on the same clock edge in synchronizing registers in the aFIFO resulting in a skipped clock edge and variable latency. While this doesn't affect overall system throughput by any measurable amount, it may corrupt a latency-sensitive system where the exact number of cycles between data transfers is part of system correctness – Condition 7 circumvents this problem.

Condition 7. (Permapaths) The intermediate clock period must be an exact multiple of the NoC clock to avoid clock drift and ensure clock edges have a consistent alignment.

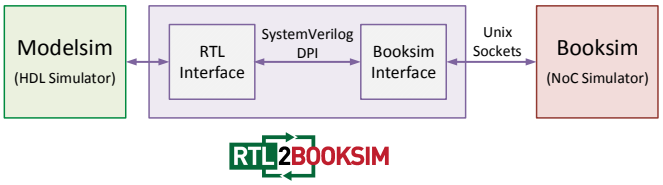


Fig. 10: `RTL2Booksim` allows the cycle-accurate simulation of an NoC within an RTL simulation in Modelsim.

5 Application Case Studies

5.1 Simulator

To measure application performance, in terms of latency and throughput, we need to perform cycle-accurate simulations of hardware designs that use an NoC. However, full register-transfer level (RTL) simulations of complex hardware circuits like NoCs are error-prone and slow. Instead of using a hardware description language (HDL) implementation of the NoC, we used a cycle-accurate software simulator of NoCs called Booksim [16]. This is advantageous because Booksim provides the same simulation cycle-accuracy, but runs faster than an HDL model of the NoC, and supports more NoC variations. Additionally, we are able to define our own packet format which greatly simplifies the interface to the NoC. Finally, it is much easier to extend Booksim with a new router architecture or special feature, making it a useful research tool in fine-tuning the NoC architecture.

Booksim conventionally simulates an NoC from a trace file that describes the movement of packets in an NoC. However, our FabricPort and applications are written in Verilog (an HDL). How do we connect our hardware Verilog components to a software simulator such as Booksim? We developed `RTL2Booksim` to interface HDL designs to Booksim; Fig. 10 shows some details of this interface.

The Booksim Interface is able to send/receive flits and credits to/from the NoC modeled by the Booksim simulator through Unix sockets. Next, there is an RTL Interface that communicates with our RTL HDL design modules. The RTL Interface communicates with the Booksim Interface through a feature of the SystemVerilog language called the direct programming interface (DPI). SystemVerilog DPI basically allows calling software functions written in C/C++ from within a SystemVerilog design file. Through these two interfaces – the Booksim Interface and the RTL interface – we can connect any hardware design to any NoC that is modeled by Booksim. `RTL2Booksim` is released as open-source and available for download at: http://www.eecg.utoronto.ca/~mohamed/rtl2booksim. The release includes push-button scripts that correctly start and end simulation for example designs using Modelsim and `RTL2Booksim`.

5.2 DDR3 Memory with IOLink

External memory interfaces, especially to DDRx memory, are some of the most important and highest bandwidth I/O interfaces on FPGAs. In this section we show how an IOLink can improve both the latency and area-utilization of external memory interfaces.

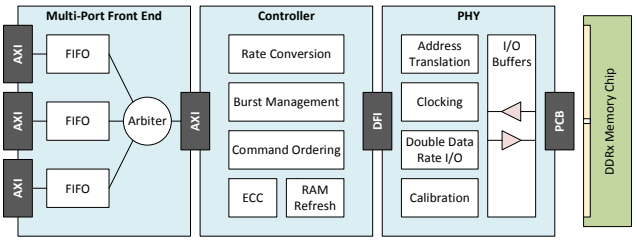


Fig. 11: Block diagram of a typical DDRx memory interface in a modern FPGA.

5.2.1 Memory Interface Components

Fig. 11 shows a typical FPGA external memory interface. The PHY is used mainly for clocking, data alignment and calibration of the clock and data delays for reliable operation, and to translate double-rate data from memory to single-rate data on the FPGA. In modern FPGAs, especially the ones that support fast memory transfers, the PHY is typically embedded in hard logic.

The memory controller (see Fig. 11) is in charge of higher-level memory interfacing. This includes regularly refreshing external memory; additionally, addresses are translated into bank, row and column components, which allows the controller to issue the correct memory access command based on the previously accessed memory word. The memory controller is sometimes implemented hard and sometimes left soft, but the trend in new devices is to harden the memory controller to provide an out-of-the-box working memory solution [17]. Some designers may want to implement their own high-performance memory controllers to exploit patterns in their memory accesses for instance, therefore, FPGA vendors also allow direct connection to the PHY, bypassing the hard memory controller. However, hard memory controllers are more efficient and much easier to use making it a more compelling option, especially as FPGAs start being used by software developers (in the context of high-level synthesis and data center computing) who do not have the expert hardware knowledge to design a custom memory controller.

The final component of a memory interface is the multi-ported front end (MPFE). This component allows access to a single external memory by multiple independent modules. It consists primarily of FIFO memory buffers to support burst transfers and arbitration logic to select among the commands bidding for memory access. The MPFE is also sometimes hardened on modern FPGAs. Beyond the MPFE, a soft bus is required to distribute memory data across the FPGA to any module that requires it.

5.2.2 Rate Conversion

One of the functions of an FPGA memory controller is rate conversion. This basically down-converts the data frequency from the high memory frequency (~1 GHz) to a lower FPGA-compatible frequency (~200 MHz). All modern memory controllers in FPGAs operate at quarter rate; meaning, the memory frequency is down-converted 4× and memory

width is parallelized eightfold[2]. Modern FPGA DDR4 memory speeds go up to 1333 MHz (in Xilinx Ultrascale+ for example [18]); at quarter rate, this translates to 333 MHz in the FPGA fabric which is challenging to achieve. Quarter-rate conversion is *necessary* to be able to use fast DDRx memory on current FPGAs – how else can we transport and process fast memory data at the modest FPGA speed? However, there are both performance and efficiency disadvantages that arise due to quarter-rate conversion in the memory controller.

Area Overhead: Down-converting frequency means up-converting data width from 128-bits (at single data rate) to 512-bits. This 4× difference increases the area utilization of the memory controller, the MPFE (including its FIFOs), and any soft bus that distributes memory data on the FPGA.

Latency Overhead: Operating at the lower frequency increases memory transfer latency. This is mainly because each quarter-rate clock cycle is much slower (4× slower) than a full-rate equivalent.

5.2.3 Proposed IOLink

We propose directly connecting an NoC link to I/O interfaces. For the external memory interface, we propose connecting an IOLink to the AXI port after the hard memory controller (see Fig. 11). We also propose implementing a memory controller that supports full-rate memory operations, even at the highest memory speeds. This topology leverages the high speed and efficiency of a full-rate controller, and avoids the costly construction of a MPFE and soft bus to transport the data. Instead, an efficient embedded NoC fulfills the function of both the MPFE and soft bus in buffering and transporting DDRx commands and data, furthermore, it does so at full-rate memory speed and lower latency.

Table 2 details the latency breakdown of a memory read transaction when fulfilled by a current typical memory interface, and an estimate of latency when an embedded NoC is connected directly to a full-rate memory controller. We use the latency of the memory chip, PHY and controller from Altera's datasheets [17]. For the MPFE, we estimate that it will take at least 2 system clock cycles[3] (equivalent to 8 memory clock cycles) to buffer data in a burst adapter and read it back out – this is a very conservative estimate on the latency of a hard MPFE. To evaluate the soft bus, we generate buses in Altera's Qsys system integration tool with different levels of pipelining. Only highly pipelined buses (3-5 stages of pipelining) can achieve timing closure for a sample 800 MHz memory speed (200 MHz at quarter rate) [6]. The round-trip latency of these buses in the absence of any traffic is 6-11 system clock cycles (depending on the level of pipelining).

To estimate the embedded NoC latency in Table 2, we used the zero-load latency from Fig. 6. The round-trip latency consists of the input FabricPort latency, the output FabricPort latency and twice the link traversal latency. At a 300 MHz fabric (system) frequency, FabricPort input latency is ~2 cycles, FabricPort output latency is 3 cycles and link traversal latency ranges between 1.5-6 cycles depending on the number of routers traversed. This adds up to a round-trip latency between 8-17 system clock cycles.

2. Width is multiplied by 8 during quarter-rate conversion because DDRx memory data operates at double rate (both positive and negative clock edges) while the FPGA logic is synchronous to either a rising or falling clock edge

3. We define a "system clock cycle" to be equivalent to the quarter-rate speed of the memory controller in our examples.

TABLE 2: Read transaction latency comparison between a typical FPGA quarter-rate memory controller, and a full-rate memory controller connected directly to an embedded NoC link. Latency is measured in full-rate memory clock cycles.

Current System		NoC-Enhanced System	
Component	Latency	Component	Latency
Memory	5-11	Memory	5-11
PHY ($\frac{1}{4}$-rate)	22-28	PHY (full-rate)	4
Controller ($\frac{1}{4}$-rate)	28	Controller (full-rate)	15
MPFE	>8	MPFE	–
Soft Bus	24-44	Hard NoC	32-68
Total	87-119	Total	56-98
Speedup = 1.2–1.6×			

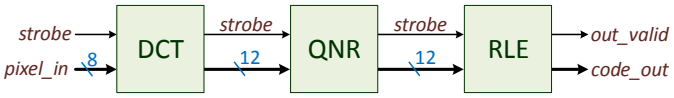


Fig. 12: Single-stream JPEG block diagram.

As Table 2 shows, the embedded NoC can improve latency by approximately 1.2–1.6× . Even though the embedded NoC has a higher round-trip latency compared to soft buses, latency improves because we use a full-rate memory controller, and avoid a MPFE. We directly transport the fast memory data using an NoC link, and only down-convert the data at a FabricPort output at the destination router where the memory data will be consumed. This undoubtedly reduces area utilization as well. More importantly, an embedded NoC avoids time-consuming timing closure iterations when connecting to an external memory interface, and improves area and power as shown in prior work [6].

5.3 JPEG Compression

We use a streaming JPEG compression design from [19]. The application consists of three modules as shown in Fig. 12; discrete cosine transform (DCT), quantizer (QNR) and run-length encoding (RLE). The single pipeline shown in Fig. 12 can accept one pixel per cycle and a data strobe indicates the start of 64 consecutive pixels forming one (8×8) block on which the algorithm operates [19]. The components of this system are therefore latency-sensitive as they rely on pixels arriving every cycle, and the modules do not respond to backpressure.

We parallelize this application by instantiating multiple (10–40) JPEG pipelines in parallel; which means that the connection width between the DCT, QNR and RLE modules varies between 130 bits and 520 bits. Parallel JPEG compression is an important data-center application as multiple images are often required to be compressed at multiple resolutions before being stored in data-center disk drives; this forms the back-end of large social networking websites and search engines. We implemented this parallel JPEG application using direct soft point-to-point links, then mapped the same design to use the embedded NoC between

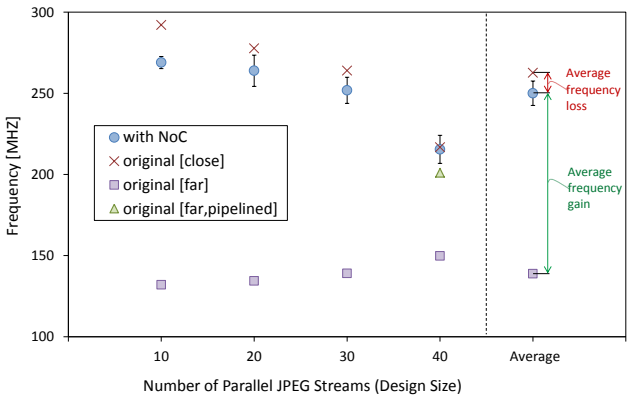


Fig. 13: Frequency of the parallel JPEG compression application with and without an NoC. The plot "with NoC" is averaged for the two cases when its modules are "close" and "far" with the standard deviation plotted as error bars. Results are averaged over 3 CAD tool seeds.

the modules using **Permapaths**. Using the `RTL2Booksim` simulator, we connected the JPEG design modules through the FabricPorts to the embedded NoC and verified functional correctness of the NoC-based JPEG. Additionally, we verified that throughput (in number of cycles) was the same for both the original and NoC versions.

5.3.1 Frequency

To model the physical design repercussions (placement, routing, critical path delay) of using an embedded NoC, we emulated embedded NoC routers on FPGAs by creating 16 design partitions in Quartus II that are of size 7×5=35 logic clusters – each one of those partitions represents an embedded hard NoC router with its FabricPorts and interface to FPGA (see Fig. 14 for the chip plan). We then connected the JPEG design modules to this emulated NoC. Additionally, we varied the physical location of the QNR and RLE modules (through location constraints) from "close" together on the FPGA chip to "far" on opposite ends of the chip.

Using location constraints, we investigated the result of a stretched critical path in an FPGA application. This could occur if the FPGA is highly utilized and it is difficult for the CAD tools to optimize the critical path as its endpoints are forced to be placed far apart, or when application modules connect to I/O interfaces and are therefore physically constrained far from one another. Fig. 13 plots the frequency of the original parallel JPEG and the NoC version. In the "close" configuration, the frequency of the original JPEG is higher than that of the NoC version by ~5%. This is because the JPEG pipeline is well-suited to the FPGA's traditional row/column interconnect. With the NoC version, the wide point-to-point links must be connected to the smaller area of an embedded router; making the placement less regular and on average slightly lengthening the critical path.

The advantage of the NoC is highlighted in the "far" configuration when the QNR and RLE modules are placed far apart thus stretching the critical path across the chip diagonal. In the NoC version, we connect to the closest NoC router as shown in Fig. 14 – on average, the frequency improved by ~80%. Whether in the "far" or "close" setups, the NoC-version's frequency only varies by ~6% as the error bars show in Fig. 13. By relying on the NoC's predictable frequency in connecting modules together, the effects of the FPGA's utilization level and the modules' physical placement constraints become localized to each module instead of being a global effect over the entire design. Modules connected through the NoC become timing-independent making for an easier CAD problem and allowing parallel compilation.

With additional design effort, a designer of the original (without NoC) system would identify the critical path and attempt to pipeline it so as to improve the design's frequency. This design→compile→repipeline cycle hurts designer productivity as it can be unpredictable and compilation could take days for a large design. We added 1–4 pipeline registers on the critical path of the original "far" JPEG with 40 streams and this improved frequency considerably, but it was still 10% worse than the NoC version. This is shown as a green triangle in Fig 13.

TABLE 3: Interconnect utilization for JPEG with 40 streams in "far" configuration. Relative difference between NoC version and the original version is reported.

Interconnect Resource		Difference	Geomean
Short	Vertical (C4)	+13.2%	+10.2%
	Horizontal (R3,R6)	+7.8%	
Long	Vertical (C14)	-47.2%	-38.6%
	Horizontal (R24)	-31.6%	

Wire naming convention: C=column, R=row, followed by number of logic clusters of wire length.

5.3.2 Interconnect Utilization

Table 3 quantifies the FPGA interconnect utilization difference for the two versions of 40-stream "far" JPEG. The NoC version reduces long wire utilization by ~40% but increases short wire utilization by ~10%. Note that long wires are scarce on FPGAs, for the Stratix V device we use, there are 25× more short wires than there are long wires. By offloading long connections onto an NoC, we conserve much of the valuable long wires for use by the application logic.

Fig. 14 shows wire utilization for the two versions of 40-stream "far" JPEG and highlights that using the NoC does not produce any routing hot spots around the embedded routers. As the heat map shows, FPGA interconnect utilization does not exceed 40% in that case. Conversely, the original version utilizes long wires heavily on the long connection between QNR→RLE, with utilization going up to 100% in hot spots at the terminals of the long connection as shown in Fig. 14.

5.4 Ethernet Switch

One of the most important and prevalent building blocks of communication networks is the Ethernet switch. The embedded NoC provides a natural back-bone for an Ethernet switch design, as it includes (1) switching and (2) buffering within the NoC routers, and (3) a built-in backpressure mechanism for flow control. Recent work has revealed that an Ethernet switch achieves significant area and performance improvements when it leverages an NoC-enhanced FPGA [20]. We describe here how such an Ethernet switch can take full

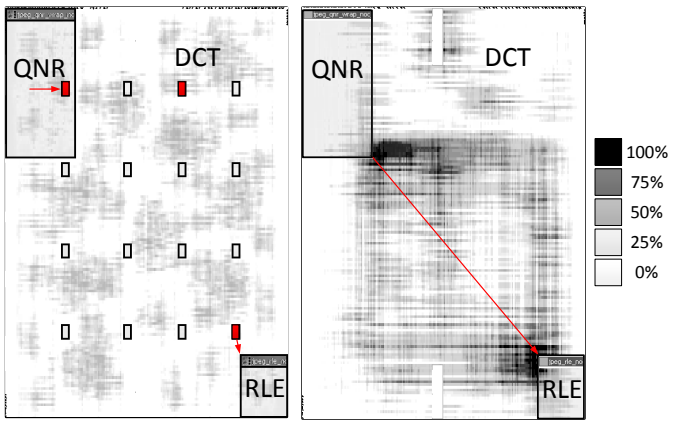


Fig. 14: Heat map showing total wire utilization for the NoC version, and only long-wire utilization for the original version of the JPEG application with 40 streams when modules are spaced out in the "far" configuration. In hot spots, utilization of scarce long wires in the original version goes up to 100%, while total wire utilization never exceeds 40% for the NoC version.

advantage of the embedded NoC, while demonstrating that it considerably outperforms the best previously proposed FPGA switch fabric design [21].

The embedded NoC is used in place of the switch's crossbar. For a 16×16 switch, each of the 16 transceiver nodes are connected to one of the 16 NoC routers via the FPGA's soft fabric. Fig. 15 shows the path between transceiver 1 and transceiver 2; in our 16×16 switch there are 256 such paths from each input to each output. On the receive path (*Rx*), Ethernet data is packed into NoC flits before being brought to the FabricPort input. The translator sets NoC control bits such that one NoC packet corresponds to one Ethernet frame. For example, a 512-byte Ethernet frame is converted into 32 NoC flits. After the NoC receives the flit from the FabricPort, it steers the flit to its destination, using dimension-order XY routing. On the transmit path (*Tx*), the NoC can output up to four flits (600 bits) from a packet in a single system clock cycle – this is demultiplexed in the output translator to the output queue width (150 bits). This demultiplexing accounts for most of the translators' area in Table 4. The translator also strips away the NoC control bits before inserting the Ethernet data into the output queue.

We synthesized the soft logic on a Stratix V device, and The design is synthesized on a Stratix V device and show the resource utilization in Table 4. Because we take advantage of the NoC's switching and buffering our switch is ~3× more area efficient than previous FPGA Ethernet switches [21] even when accounting for the complete embedded NoC area.

Two important performance metrics for Ethernet switch design are bandwidth and latency [22]. The bandwidth of our NoC-based Ethernet switch is limited by the supported bandwidth of the embedded NoC. As described in Section 2, the NoC's links have a bandwidth capacity of 22.5 GB/s (180 Gb/s). Since some of this bandwidth is used to transport packet control information, the NoC's links can support up to 153.6 Gb/s of Ethernet data. Analysis of the worst case

TABLE 4: Hardware cost breakdown of an NoC-based 10-Gb Ethernet switch on a Stratix V device. NoC area* is reported in equivalent ALM area.

	10GbE MACs	I/O Queues	Trans--lators	NoC Switch	**Total**
ALMs	24000	3707	3504	5280*	**36491**
M20Ks	0	192	0	0	**192**

traffic in a 16-node mesh shows that the NoC can support a line rate of one third its link capacity, i.e. 51.2 Gb/s [20]. While previous work on FPGA switch design has achieved up to 160 Gb/s of aggregate bandwidth [21], our switch design can achieve 51.2×16 = 819.2 Gb/s by leveraging the embedded NoC. We have therefore implemented a programmable Ethernet switch with 16 inputs/outputs that is capable of either 10 Gb/s, 25 Gb/s or 40 Gb/s – three widely used Ethernet standards.

The average latency of our Ethernet switch design is measured using the `RTL2Booksim` simulator. An ON/OFF injection process is used to model bursty, uniform random traffic, with a fixed Ethernet frame size of 512 bytes (as was used in [21]). Latency is measured as the time between a packet head being injected into the input queue and it arriving out of the output queue. Fig. 16 plots the latency of our Ethernet switch at its supported line rates of 10 Gb/s, 25 Gb/s and 40 Gb/s. Surprisingly, the latency of a 512 byte packet improves at higher line rates. This is because a higher line rate means a faster rate of injecting NoC flits, and the NoC can handle the extra switching without a large latency penalty thus resulting in an improved overall latency. No matter what the injection bandwidth, the NoC-based switch considerably outperforms the Dai/Zhu switch [21] for all injection rates. By supporting these high line rates, our results show that an embedded NoC can push FPGAs into new communication network markets that are currently dominated by ASICs.

5.5 Packet Processor

In recent years there has been a surge in demand on computer networks, causing a rapid evolution in network protocols and functionality. Programmable network hardware has hence become highly desirable [23], [24], as it can provide both the flexibility to evolve and the capacity to support the latest bandwidth demands. Prior work has demonstrated two ways to implement a flexible and high performing packet processor: the PP packet processor [25], built from an FPGA, and the RMT packet processor [26], built from ASIC technology. Although both provide varying trade-offs between flexibility and performance, we believe a better trade-off can be reached by using a new packet processor design built from the NoC-enhanced FPGA.

Unlike previously proposed programmable packet processors that use an OpenFlow-like cascade of programmable "flow tables" [24], our "NoC Packet Processor" (NoC-PP [27]) uses a modular design style, where various modules are implemented in the FPGA fabric, each dedicated to processing a single network protocol (e.g. Ethernet, IPv4, etc.). Packets are switched between these protcol processing modules via the embedded NoC. The flexibility of the FPGA fabric allows the

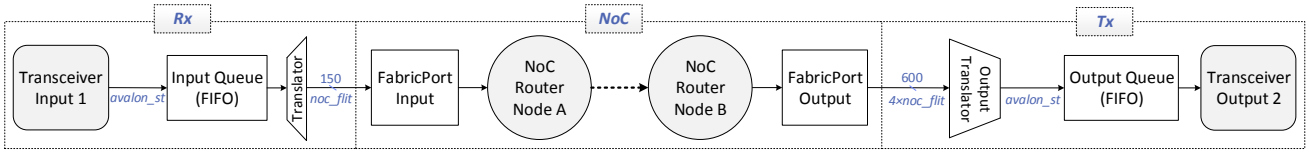


Fig. 15: Functional block diagram of one path through our NoC Ethernet switch.

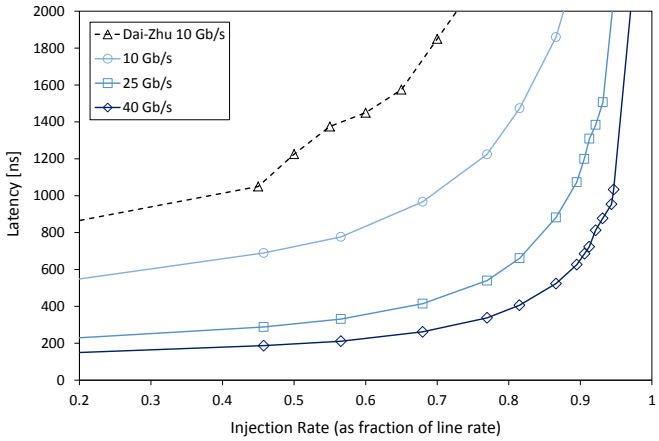


Fig. 16: Latency vs. injection rate of the NoC-based Ethernet switch design given line rates of 10, 25, and 40 Gb/s, and compared to the Dai/Zhu 16×16 10 Gb/s FPGA switch fabric design [21]. Our switch queues and Dai/Zhu's switch queues are of size 60kb and 16kb, respectively.

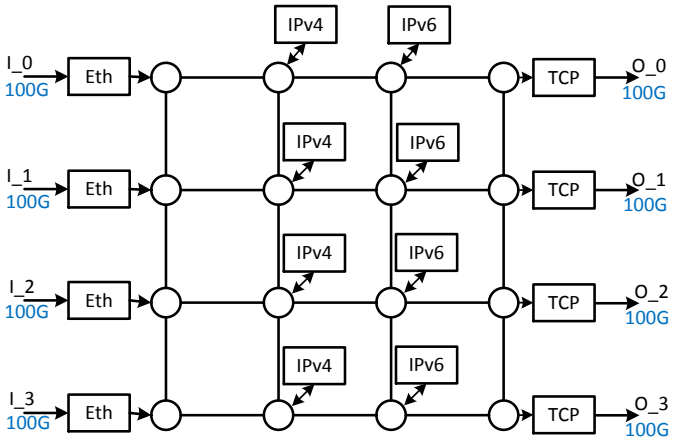


Fig. 17: The NoC-PP design for an Ethernet/VLAN/IPv4/IPv6/TCP packet processor (Eth=Ethernet+VLAN).

TABLE 5: Comparison of the NoC-PP and PP architectures

Application	Architecture	Resource Utilization (% FPGA)	Latency (ns)	Throughput (Gb/s)
JustEth	NoC-PP	3.6%	79	400
	PP [25]	11.6%	293	343
TcpIp4Ip6	NoC-PP	9.4%	200	400
	PP [25]	15.6%	309	325

modules to be fully customized and later updated, as existing protocols are enhanced and new protocols are added. The embedded NoC provides an efficient interconnect that can support switching packets between the modules at modern network bandwidths.

To evaluate this design, we implemented a packet processor that supports processing of several common network protocols: Ethernet, VLAN, IPv4, IPv6, and TCP. Each packet going through the processor will visit a processing module for each protocol found in its header. The processing modules are designed with a data path width of 512 bits running at 200 MHz, providing an overall processing throughput of 100 Gb/s. In order to support higher network bandwidths, several copies of the processing modules are instantiated in the fabric as desired (in this case four instantiations to support 400G, see Figure 17). Having the generality of the FPGA fabric provides an important advantage to FPGA-based packet processing; whereas the ASIC-based RMT design [26] provides some flexibility, it is still limited to what is made available upon chip fabrication [27].

We measure the hardware cost and performance of the NoC-PP design and compare it to the PP design [25], another efficient FPGA-based packet processor. We compare to two versions of the PP design: (1) "JustEth", which only performs parsing on the Ethernet header, and (2) "TcpIp4andIp6", which performs parsing on Ethernet, IPv4, IPv6 and TCP [25]. Table 5 contains hardware cost and performance results of the NoC-PP and PP designs, with hardware cost measured using resource utilization as a percentage of an Altera Stratix V-GS FPGA. Overall, the NoC-PP proves to be more resource efficient and achieves better performance compared to the PP architecture while providing the same degree of hardware flexibility via the FPGA fabric. For the smaller application (JustEth), the NoC-PP design is 3.2× more efficient, whereas for the larger application (TcpIp4Ip6), it is 1.7× more efficient. NoC-PP also reduces latency by 3.7× and 1.5× compared to PP for JustEth and TcpIp4Ip6, respectively.

It is also important to determine what brings these efficiencies to NoC-PP; is it the new module-based packet processor architecture, the introduction of the hard NoC, or a synergistic fusion of the two? To answer this question, we began by replacing the hard NoC in our design with an equivalent soft NoC, and separately quantified the cost of the NoC and the processing modules. We also built another iteration of our design using customized soft crossbars such that only modules that need to communicate are connected. As can be seen in Figure 18, the costs of the soft NoC and the soft crossbar are 29× and 11× greater than that of the hard NoC, respectively. The significantly higher cost of building NoC-PP's interconnection network out of the reconfigurable FPGA fabric is due to the fact it runs at a considerably lower clock frequency compared to the hard NoC and must

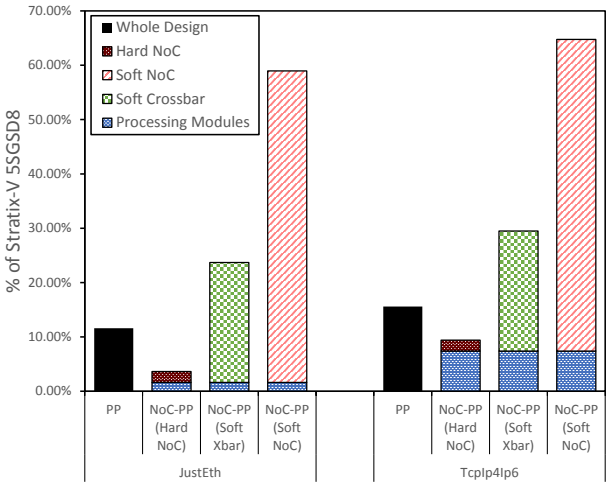


Fig. 18: Area breakdown of NoC-PP when using a hard NoC, a soft NoC or a soft custom crossbar (Xbar).

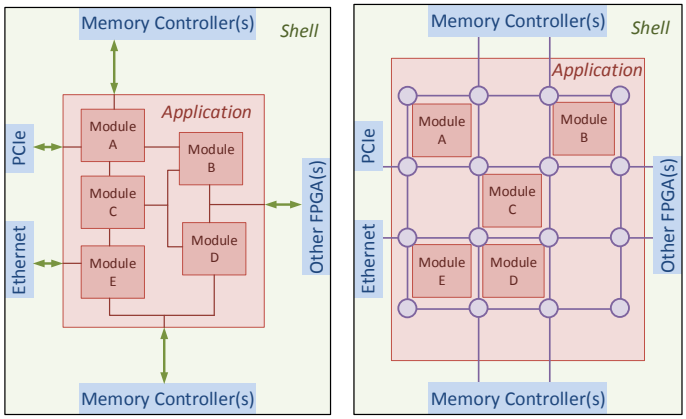


Fig. 19: A sample FPGA compute accelerator consists of a shell and the application logic.

therefore use wide datapaths to transport the high bandwidth data. Switching between these wide datapaths requires large multiplexers and wide buffers that consume high amounts of resources. The design therefore achieves significant savings by hardening this interconnect in the embedded NoC.

Since the processing modules form a small fraction of the design cost when using a soft interconnect, NoC-PP can therefore achieve significant overall savings when replacing the soft interconnect with a hard NoC. On the other hand, the PP design uses a feed-forward design style. Rather than switching between protocol modules, PP uses tables containing "microcode" entries for all possible protocols that must be processed at that stage [25]. Thus, no wide multiplexing exists in the design that can be efficiently replaced by a hard NoC. The logic and memory within each stage form the majority PP's hardware cost, which would not change if a hard NoC was introduced. We therefore conclude that the efficiencies from NoC-PP stem from a synergistic fusion of using the hard NoC with our module-based packet processor architecture.

5.6 Embedded NoC for FPGA Computing

In this future-looking section, we discuss how an embedded NoC could improve FPGA utility in emerging but important areas such as data center compute acceleration [1] or the OpenCL compute model [28]. FPGA accelerators consist primarily of two components; the application logic itself, and the *shell* which implements the communication infrastructure. Fig. 19 illustrates the implementation of a compute accelerator, both with and without an embedded NoC.

As Fig. 19 shows, a shell consists primarily of a system-level interconnect that connects an application to the host CPU, I/O and memory interfaces, and other FPGAs [1], [28]. Built out of soft buses, Microsoft's implementation of this shell occupies approximately one quarter (23% area) of a large Stratix V FPGA device [1]. We believe this area overhead can be reduced if an NoC (with an area of ~1.3%) is leveraged to interconnect modules in the shell and accelerator – previous work has already shown that an NoC uses less area and power than soft buses when connecting a system to DDRx memory [6]. Besides the area overhead, it is challenging to meet the timing constraints of the many fast I/O interfaces to which the shell connects – consequently designers typically *lock down* the placement and routing of the shell once timing closure is attained, and present standard fixed-location interfaces to the FPGA application logic [1]. Conversely, an NoC with direct IOLinks to external interfaces can significantly ease timing closure. Furthermore, any NoC router can be used to connect application modules to the shell instead of fixed-location interfaces – this would relax the placement and routing constraints of the application and likely improve its performance. As the JPEG application case study showed, overall application frequency can also be much more predictable with an NoC. This predictability is especially important when high-level languages (such as OpenCL) are used, and the designer has no direct way to improve operating frequency.

In such compute accelerators, partial reconfiguration is quickly becoming an important feature [1]. Using partial reconfiguration, the application could either be repaired or replaced without powering down the FPGA, or the data center node in which it lies. To successfully connect a partially reconfigured module, current accelerator shells must provide fixed interfaces and interconnect to the *superset* of the modules that could be configured on the FPGA. This method could be wasteful and complex, and it is often difficult to predict exactly what will be reconfigured in the future. Instead, we propose using the NoC FabricPorts as a standard, yet flexible interface to partially reconfigured modules. Even though this might constrain module placement to be close to one of the fixed-location FabricPorts, it avoids the need to explicitly provision a soft bus interface and interconnect for all partially reconfigured modules. Any module connected through a FabricPort will be able to communicate with the rest of the application and I/Os through the embedded NoC, reducing the need for additional interconnect or glue logic for partial reconfiguration. Our packet processor, NoC-PP in Section 5.6, could similarly benefit from partial reconfiguration – the processing modules can be partially reconfigured to update or change the networking protocol.

6 Conclusion

We proposed augmenting FPGAs with an embedded NoC and focused on how to use the NoC for transporting data in FPGA applications of different design styles. The FabricPort is a flexible interface between the embedded NoC and the FPGA's core; it can bridge any fabric frequency and data width up to 600 bits to the faster but narrower NoC at 1.2 GHz and 150 bits. To connect to I/O interfaces, we proposed using direct IOLinks and have shown that they can reduce DDR3 access latency by 1.2–1.6× . We also discussed the conditions under which FPGA design styles can be correctly implemented using an embedded NoC. Next, we presented a `RTL2Booksim`: a simulator that enables the cycle-accurate co-simulation of a software NoC simulator and hardware RTL designs. Our application case studies showed that JPEG image compression frequency can be improved by 10–80× and the embedded NoC avoids wiring hotspots and reduces the use of scarce long wires by 40% at the expense of a 10% increase of the much more plentiful short wires. We also showed that high-bandwidth Ethernet switches can be efficiently constructed on the FPGA; by leveraging an embedded NoC we created an 819 Gb/s programmable Ethernet switch – a major improvement over the 160 Gb/s achieved by prior work in a traditional FPGA. Finally, we presented a new way of implementing packet processors leveraging our proposed embedded NoC and showed that it is 1.7–3.2× more area efficient and 1.5–3.7× lower latency compared to previous work.

References

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014, pp. 13–24.
[2] H. Song and J. W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection Using FPGA," in *FPGA*, 2005, pp. 238–245.
[3] M. Langhammer and B. Pasca, "Floating-Point DSP Block Architecture for FPGAs," in *FPGA*. ACM, 2015, pp. 117–125.
[4] M. S. Abdelfattah and V. Betz, "Power Analysis of Embedded NoCs on FPGAs and Comparison With Custom Buses," *TVLSI*, vol. 24, no. 1, pp. 165–177, 2016.
[5] R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, 2001.
[6] M. S. Abdelfattah and V. Betz, "The Case for Embedded Networks-on-Chip on Field-Programmable Gate Arrays," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, 2014.
[7] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, "The Stratix™10 Highly Pipelined FPGA Architecture." ACM, 2016, pp. 159–168.
[8] D. U. Becker, "Efficient Microarchitecture for Network on Chip Routers," Ph.D. dissertation, Stanford University, 2012.
[9] Xilinx Inc. (2009-2014) Virtex-5,6,7 Family Overview.
[10] M. S. Abdelfattah. FPGA NoC Designer. [Online]. Available: www.eecg.utoronto.ca/~mohamed/noc_designer.html
[11] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Boston, MA: Morgan Kaufmann Publishers, 2004.
[12] M. S. Abdelfattah and V. Betz, "Networks-on-Chip for FPGAs: Hard, Soft or Mixed?" *TRETS*, vol. 7, no. 3, pp. 20:1–20:22, 2014.
[13] Altera Corp. (2014) Video and Image Processing Suite.
[14] D. J. Sorin *et al.*, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.
[15] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test*, vol. 22, no. 5, 2005.
[16] N. Jiang *et al.*, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013, pp. 86–96.
[17] *External Memory Interface Handbook*, Altera Corp., 2015.
[18] *UltraScale Architecture FPGAs Memory IP*, Xilinx Inc, 2015.
[19] A. Henson and R. Herveille. (2008) Video Compression Systems. [Online]. Available: www.opencores.org/project,video_systems
[20] A. Bitar *et al.*, "Efficient and programmable Ethernet switching with a NoC-enhanced FPGA," in *ANCS*, 2014.
[21] Z. Dai and J. Zhu, "Saturating the Transceiver BW: Switch Fabric Design on FPGAs," in *FPGA*, 2012, pp. 67–75.
[22] I. Elhanany *et al.*, "The network processing forum switch fabric benchmark specifications: An overview," *IEEE Network*, vol. 19, no. 2, pp. 5–9, 2005.
[23] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.
[24] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM*, vol. 38, no. 2, pp. 69–74, 2008.
[25] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *ANCS*, 2011, pp. 12–23.
[26] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *SIGCOMM*, vol. 43, no. 4. ACM, 2013, pp. 99–110.
[27] A. Bitar, M. Abdelfattah, and V. Betz, "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *FPT*, 2015.
[28] D. Singh, "Implementing FPGA Design with the OpenCL Standard," Altera Corp., Tech. Rep., Nov. 2013.

Mohamed S. Abdelfattah received the BSc degree in ECE from the German University in Cairo in 2009, and the MSc degree in ECE from the University of Stuttgart in 2011. He is currently pursuing the PhD degree in ECE from the University of Toronto, Canada, where he is researching new communication architectures and CAD tools for FPGAs. His research interests include FPGA architecture and CAD, on-chip communication, high-level synthesis, and datacenter acceleration. Mr. Abdelfattah is the recipient of many scholarships and awards, most notably the Vanier Canada Graduate Scholarship and two best paper awards at the FPL 2013 and FPGA 2015 conferences.

Andrew Bitar received the BASc degree in ECE from the University of Ottawa in 2013, and the MASc degree in ECE from the University of Toronto in 2015. His research interests include FPGA acceleration of datacenter and networking applications, as well as high-level synthesis. Mr. Bitar is the recipient of several scholarships and awards, including the NSERC Canada Graduate Scholarship and the best paper award at the FPGA 2015 conference. He has been a Design Engineer working on High-Level Design in the Programmable Solutions Group at Intel since 2015.

Vaughn Betz received the BSc degree in EE from the University of Manitoba in 1991, the MS degree in ECE from the University of Illinois at Urbana-Champaign in 1993, and the PhD degree in ECE from the University of Toronto in 1998. Dr. Betz was a co-founder of Right Track CAD in 1998 and its VP of Engineering until its acquisition by Altera in 2000. He held various roles at Altera from 2000 to 2011, ultimately as Senior Director of Software Engineering. He joined the University of Toronto as an Associate Professor in 2011 and holds the NSERC/Altera Chair in Programmable Silicon; his research covers FPGA architecture, CAD and FPGA-based computation.