Take the Highway: Design for Embedded NoCs on FPGAs

Mohamed S. Abdelfattah, Andrew Bitar, Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
{mohamed, bitar, vaughn}@eecg.utoronto.ca

ABSTRACT

We explore the addition of a fast embedded network-on-chip (NoC) to augment the FPGA's existing wires and switches, and help interconnect large applications. A flexible interface between the FPGA fabric and the embedded NoC allows modules of varying widths and frequencies to transport data over the NoC. We study both latency-insensitive and latency-sensitive design styles and present the constraints for implementing each type of communication on the embedded NoC. 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 on FPGAs.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)

1. INTRODUCTION

Field-programmable gate-arrays (FPGAs) are increasing in both capacity and heterogeneity. Over the past two decades, FPGAs have evolved from a chip with thousands of logic elements (and not much else) to a much larger chip that has millions of logic elements, embedded memory, multipliers, processors, memory controllers, PCIe controllers and high-speed transceivers [26]. This incredible increase in size and functionality has pushed FPGAs into new markets and larger and more complex systems [24].

Both the FPGA's logic and I/Os have had efficient embedded units added to enhance their performance; however, the FPGA's interconnect is still basically the same. Using a combination of wire segments and multiplexers, a single-bit connection can be made between any two points on the FPGA chip. While this traditional interconnect is very flexible, it is becoming ever-more challenging to use in connecting large systems. Wire-speed is scaling poorly compared to transistor speed [19], and a larger FPGA device means that a connection often consists of multiple wire segments and multiplexers thus increasing overall delay. This makes it difficult to estimate the delay of a connection before placement and routing, forcing FPGA designers to wait until design compilation is completed, then identify the critical path and manually add pipeline registers in an attempt to improve frequency – a time-consuming process. Furthermore, the high bandwidth of embedded I/O interfaces requires fast and very wide connections that distribute data across the whole chip. This utilizes much FPGA logic and a multitude of its single-bit wires and multiplexers; consequently, it is difficult to run these wide connections fast enough to satisfy the stringent delay constraints of interfaces like DDR3.

FPGA '15, February 22–24 2015, Monterey, CA, USA

http://dx.doi.org/10.1145/2684746.2689074 ...$15.00.

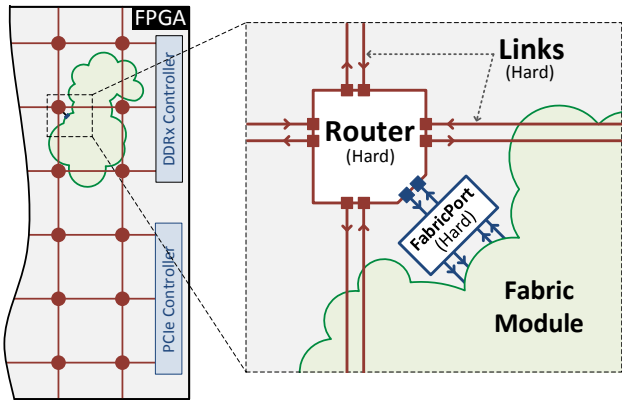


Figure 1: Embedded hard NoC connects to the FPGA fabric and hard I/O interfaces.

System-level interconnect has been proposed to augment the FPGA's bit-level interconnect to better integrate large systems. Some have suggested the use of bus-based FPGA interconnect to save area [27], while others have investigated embedded NoCs [5, 15, 17]. In this work we focus on the latter; specifically, how to interface the FPGA fabric to an embedded NoC, and how to use an embedded NoC for different design styles that are common to FPGAs. Previous work has investigated how to use an embedded NoC to create a multiprocessor-like memory abstraction for FPGAs [9]. In contrast, we focus on *adapting* an embedded NoC to the currently used FPGA design styles. 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. Investigate the requirements of mapping the communication of different design styles (latency-insensitive and latency-sensitive) onto an embedded NoC.
3. Analyze latency-sensitive parallel JPEG compression both with and without an embedded NoC.
4. Design an Ethernet switch capable of 819 Gb/s using the embedded NoC; 5× more switching than previously demonstrated on FPGAs.

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.
*Percentage of core area of a large Stratix V FPGA.

2. EMBEDDED HARD NOC

Before presenting our embedded NoC, we define some of the NoC terminology [12] that may be unfamiliar to the reader:

- Flit: The smallest unit of data that can be transported on the NoC; it is equivalent to the NoC link width.
- Packet: One or more related flits that together form a logical meaning.
- Virtual channels (VCs): Separate FIFO buffers at a NoC router input port; if we use 2 VCs in our NoC, then each router input can store incoming flits in one of two possible FIFO buffers.
- Credit-based flow control: A backpressure mechanism in which each NoC router keeps track of the number of available buffer spaces (credits) downstream, and only sends a flit downstream if it has available credits.

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. Fig. 1 displays a high-level view of an NoC embedded on an FPGA. We base our router design on a state-of-the-art full-featured packet-switched router [10].

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/Os 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, or to one of the masters accessing it 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] [1]. 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 summarizes the NoC parameters and properties. We use 2 VCs in our NoC. Previous work has shown that a second VC reduces congestion by ~30% [3]. We also leverage VCs to avoid deadlock, and merge data streams as we discuss in Sections 3 and 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 (see Section 4.3.1). 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 [4]. 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) [26].

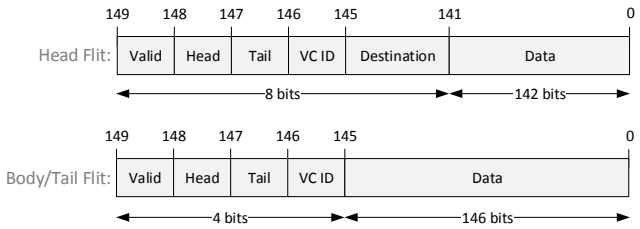


Figure 2: NoC packets consist of a head flit and zero-or-more body flits. The figure shows flits for a 16-node 150-bit-width NoC with 2 VCs. Each flit has control data to indicate whether this flit is valid, and if it is the head or tail flit (or both for a 1-flit packet). Additionally each flit must have the VC number to which it is assigned and a head flit must contain the destination address.

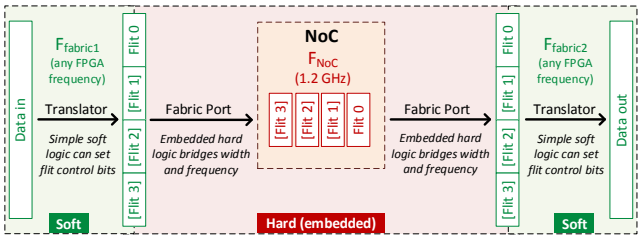


Figure 3: Data on the FPGA with any protocol can be translated into NoC flits using application-dependent soft logic (translator). A FabricPort then adapts width (1-4 flit width on fabric side and 1 flit width on NoC) and frequency (any frequency on fabric side and 1.2 GHz on NoC side) to inject flits into the NoC.

3. FABRICPORT: INTERFACE BETWEEN FPGA AND NOC

3.1 Packet Format

Fig. 2 shows the format of flits on the NoC; each flit is 150 bits making flit width and NoC link width equivalent (as most on-chip networks do) [12]. One flit is the smallest unit that can be sent over the NoC, indicating that the NoC will be used for coarse-grained wide datapath transfers. This packet format puts no restriction on the number of flits that form a packet; each flit has two bits for "head" and "tail" to indicate the flit at the start of a packet, and the flit at the end of a packet. The VC identifier is required for proper virtual-channel flow control, and finally, the head flit must also contain the destination address so that the NoC knows where to send the packet. The remaining bits are data, making the control overhead quite small in comparison; for a 4-flit packet, control bits make up 3% of transported data.

3.2 FabricPort Functionality

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. 3 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 neces-

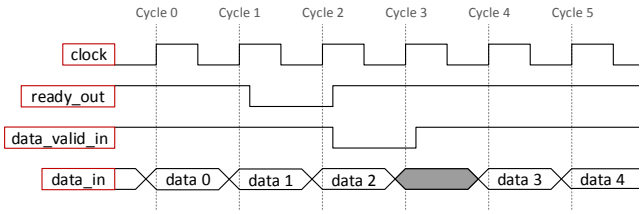


Figure 4: Waveform of ready/valid signals between soft module → FabricPort input, or FabricPort output → soft module. After "ready" signal becomes low, the receiver must accept one more cycle of valid data (data 2) after which the sender will have processed the "ready" signal and stopped sending more valid data.

sary 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 fabric 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 [2, 4]. 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 exact number of cycles when the fabric/NoC isn't ready to send/receive data (as Fig. 4 shows). 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**: Send/receive packets of different lengths.
5. **Backpressure Translation**: Convert the NoC's credit-based flow-control system into the more FPGA-familiar ready/valid signals.

3.3 FabricPort Circuitry

3.3.1 *FabricPort Input: Fabric→NoC*

Fig. 5 shows a schematic of the FabricPort with important control signals annotated. The FabricPort input (Fig. 5a) 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. To clarify this ready-valid behavior, example waveforms are illustrated in Fig. 4. 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.

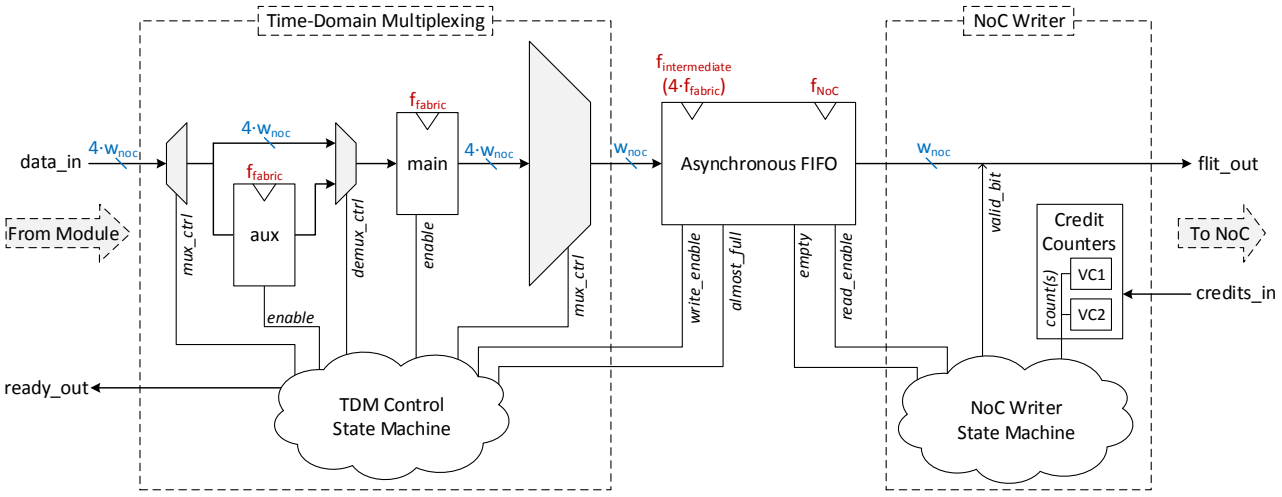
The final component in the FabricPort input is the "NoC Writer". This unit reads flits from the aFIFO and writes them to the downstream NoC router. The NoC Writer keeps track of the number of credits in the downstream router to interface to the credit-based backpressure system in the embedded NoC, and 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 [12].

3.3.2 *FabricPort Output: NoC→Fabric*

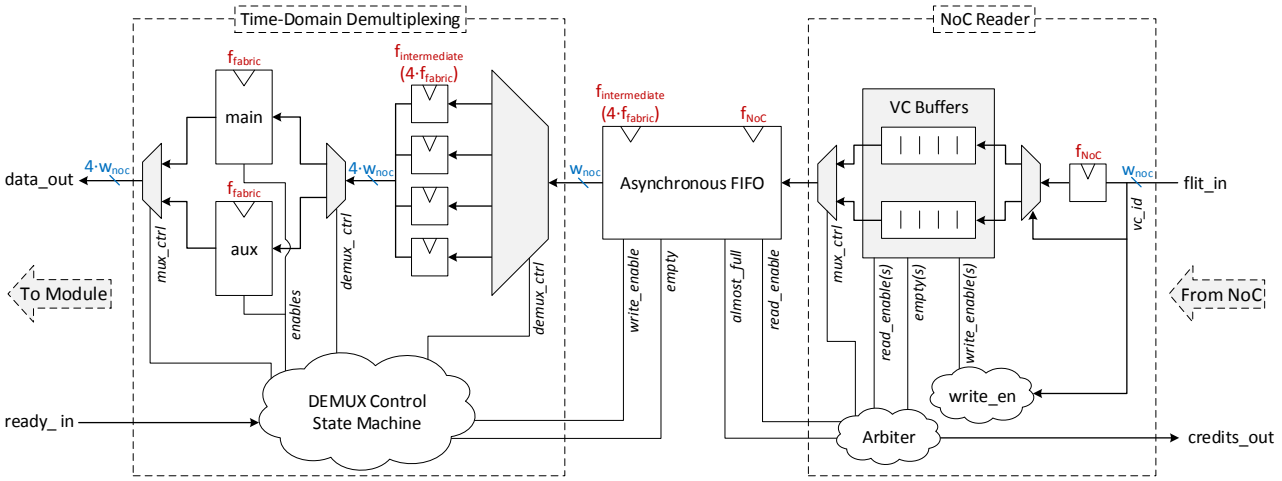
Fig. 5b details a FabricPort output; the connection from an NoC output port to the input of a module on the FPGA fabric. Following the diagram from right to left: the first component is the "NoC Reader". This unit is responsible for reading flits from an NoC router output port and writing to the aFIFO. Note that separate FIFO queues must be kept for each VC; this is very important as it avoids scrambling data from two packets. Fig. 6 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. 6). 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.



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



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

Figure 5: 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.

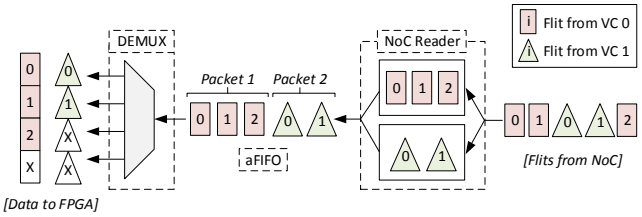


Figure 6: "NoC Reader" sorts flits from each VC into a separate queue thereby ensuring that flits of each packet are contiguous. The DEMUX then packs up to four flits together and writes them to the wide output port but never mixes flits of two packets.

3.4 FabricPort Discussion

3.4.1 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. 3) 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. 6 shows. To overcome this limitation, we create a *combine-data* mode as shown in Fig. 7. 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.*

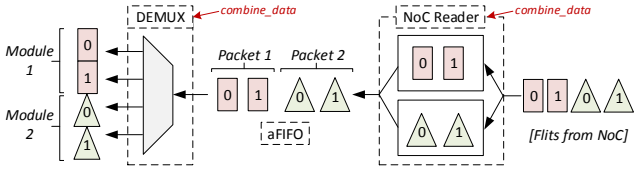


Figure 7: FabricPort output merging two packets from separate VCs in *combine-data* mode, to be able to output data for two modules in the same clock cycle.

Note that we are limited to the merging of two packets with 2 VCs but we can merge up to four 1-flit packets if we increase the number of VCs to four in the embedded NoC.

3.4.2 *Frequency and Latency*

Fig. 8 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. Careful inspection of Fig. 8 reveals that the FabricPort input always has a fixed latency for a given frequency, while the latency of the FabricPort output varies by one cycle sometimes – this is an artifact of having to wait for the *next* fabric (slow) clock cycle on which we can output data in the DEMUX unit.

4. FPGA-DICTATED NOC DESIGN

Fig. 9 shows the two possibilities of synchronous design styles, as well as two communication protocols that are common in FPGA designs. In a latency-insensitive system, the design consists of *patient* modules that can be stalled, thus allowing the interconnect between those modules to have arbitrary delay [8]. Latency-sensitive design, on the other hand, does not tolerate variable latency on its connections, and assumes that its interconnect always has a fixed latency. In this section we investigate how to map applications that belong to either design style (and any communication protocol) onto the NoC; Fig. 10 illustrates this. We are effectively augmenting the FPGA with a wide stallable network of buffered interconnect that can do flexible switching – how can we best leverage that new interconnection resource for different design styles? And can this embedded NoC be used for both latency insensitive/sensitive design styles, and both communication protocols?

4.1 Packet Ordering and Dependencies

4.1.1 *Ordering*

Packet-switched NoCs like the one we are using were originally built for chip multiprocessors (CMPs). CMPs only perform **memory-mapped** 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, memory-mapped 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

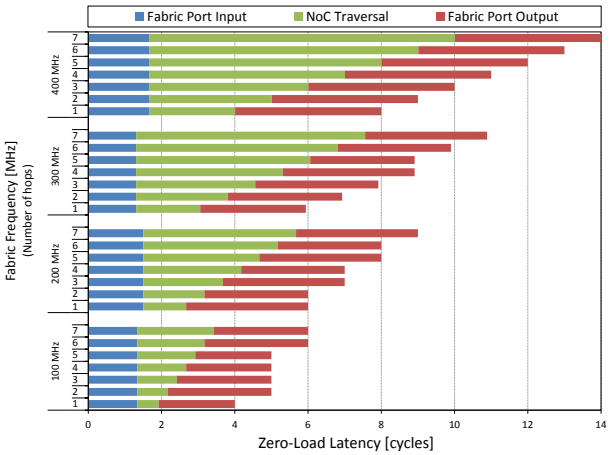


Figure 8: Zero-load latency of the embedded NoC (including FabricPorts) at different fabric frequencies. Latency is reported as the number of cycles at each frequency. The number of hops varies from 1 hop (minimum) to 7 hops (maximum – chip diagonal).

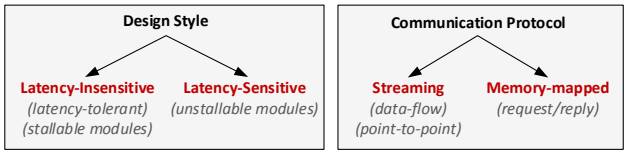


Figure 9: Design styles and communication protocols.

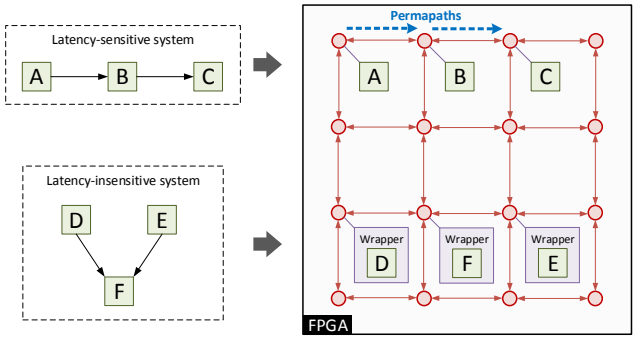
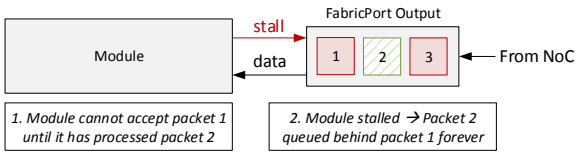


Figure 10: Mapping latency-sensitive and latency-insensitive systems onto an embedded NoC. We reserve *Permapaths* on the NoC to guarantee a fixed latency and perfect throughput for a latency-sensitive application. For latency-insensitive systems, modules must be encapsulated with wrappers to add stall functionality.

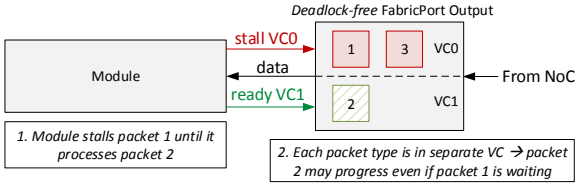
bandwidth data suitable for our proposed NoC. Additionally, FPGAs are very good at implementing **streaming**, or dataflow applications such as packet switching, video processing, compression and encryption. These streams of data are also prime targets for using our high-bandwidth embedded NoC. Crucially, neither memory-mapped 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.*



(a) Standard FabricPort output.



(b) Deadlock-free FabricPort output.

Figure 11: Deadlock can occur if a dependency exists between two message types going to the same port. By using separate VCs for each message type, this deadlock can be broken thus allowing two dependent message types to share a FabricPort output.

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 routing algorithms, in which routes 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**.*

Deterministic routing algorithms such as dimension-ordered routing [12] fulfill Condition 2 as they always select the same route for packets on 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. 7), 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** must use the same VC.*

To do this in NoC routers is simple. Normally, a packet may change VCs at every router hop – VC selection is done in a VC allocator [12]. We replace this VC allocator with a lightweight VC *facilitator* that cannot switch a packet between VCs; instead, it inspects a packet's input VC and stalls that packet until the downstream VC buffer is available. At the same time, other connections may use other VCs in that router thus taking advantage of multiple VCs.

4.1.2 Dependencies and Deadlock

Two *message types* may not share a standard FabricPort output (Fig. 5b) 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 [6]. An IP core may not be able to process the data messages correctly until it receives a control message.

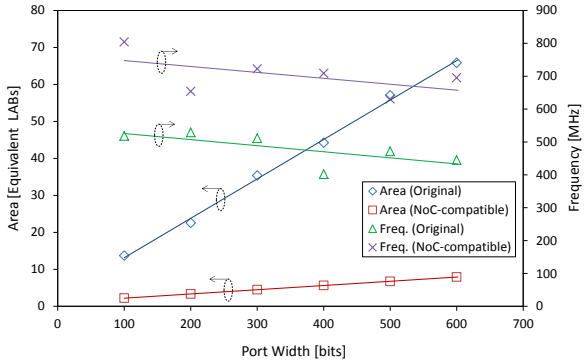


Figure 12: Area and frequency of latency-insensitive wrappers from [23] (original), and optimized wrappers that take advantage of NoC buffering (NoC-compatible).

Consider the deadlock scenario in Fig. 11a. 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 in a different VC [25]. Additionally, we created a deadlock-free FabricPort output that maintains separate paths for each VC – this means we duplicate the aFIFO and DEMUX units for each VC we have. There are now two separate "ready" signals; one for each VC, but there is still only one data bus feeding the module. The module can therefore *either* read from VC0 or VC1. Fig. 11b 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.*

4.2 Latency-Insensitive Design with NoC

Latency-insensitive design is a design methodology that decouples design modules from their interconnect by forcing each module to be *patient*; that is, to tolerate variable latency on its inputs [8]. This is typically done by encapsulating design modules with wrappers that can stall a module until its input data arrives. This means that a design remains functionally correct, by construction, regardless of the latency of data arriving at each module. The consequence of this latency tolerance is that a CAD tool can automatically add pipeline stages (called *relay stations*) invisibly to the circuit designer, late in the design compilation and thus improve frequency without extra effort from the designer [8].

Our embedded NoC is effectively a form of latency-insensitive interconnect; it is heavily pipelined and buffered and supports stalling. We can therefore leverage such an NoC to interconnect patient modules of a latency-insensitive system as illustrated in Fig. 10. Furthermore, we no longer need to add relay stations on connections that are mapped to NoC links, avoiding their overhead.

Previous work that investigated the overhead of latency-insensitive design on FPGAs used FIFOs at the inputs of modules in the stall-wrappers to avoid throughput degradation whenever a stall occurs [23]. When the interconnect is an embedded NoC; however, we already have sufficient buffering in the NoC itself (and the FabricPorts) to avoid this throughput degradation, thus allowing us to replace this FIFO – which is a major portion of the wrapper area – by a single stage of registers. We compare the area and frequency of the original latency-insensitive wrappers evaluated in [23], and the NoC-compatible wrappers in Fig. 12 for wrappers that support one input and one output and a width between

100 bits and 600 bits. As Fig. 12 shows, the lightweight NoC-compatible wrappers are 87% smaller and 47% faster.

We envision a future latency-insensitive design flow targeting embedded NoCs on FPGAs. Given a set of modules that make up an application, they would first be encapsulated with wrappers, then mapped onto an NoC such that performance of the system is maximized.

4.3 Latency-Sensitive Design with NoC (Permapaths)

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 [16]. 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 packet-switched embedded NoC.

4.3.1 Latency and Throughput Guarantees

To ensure that the NoC doesn't stall due to unavailable buffering, we size NoC buffers for maximum throughput, so that we never stall while waiting for backpressure signals within the NoC. This is well-studied in the literature and is done by sizing our router buffers to cover the *credit round-trip latency* [12] – for our system, a buffer depth of 10 suffices.

Fig. 13 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. In fact, the NoC connection acts as a simple pipelined wire; the number of pipeline stages are equivalent to the zero-load latency of an NoC path; however, it is irrelevant because 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. 13 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 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.

4.4 Multicast, Reconvergence and Feedback

A complex FPGA application may include multicast, reconvergence and feedback as shown in Fig. 14 – we discuss these aspects briefly here but leave the in-depth analysis for future work. Prior NoC research has shown that packet-switched routers can be augmented with multicast capability at very low area overhead [14]. As for reconvergence, the two branches of a reconvergent path may have different latencies on the embedded NoC with different implications for latency-sensitive and latency-insensitive systems. A latency-sensitive system may become functionally incorrect in that case; the designer must therefore ensure that the paths are balanced. For a latency-insensitive system functional correctness is guaranteed but throughput degradation may occur if latencies of the two paths differ by a large amount; prior work has investigated path balancing for latency-insensitive systems [22]. Balancing can be done by selecting two paths of the same length through the NoC (hence same latency) and using registers in the FPGA fabric for fine-grained latency adjustment. Feedback paths are also tricky to implement on embedded NoCs; this stems from the fact that these connections are typically latency-critical and require very low latency so as not to impede throughput.

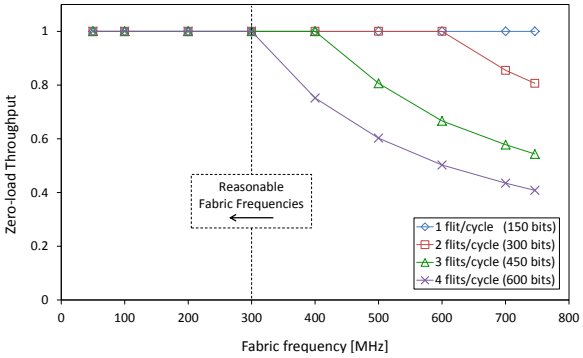


Figure 13: Zero-load throughput of embedded NoC path between any two nodes, normalized to sent data. A throughput of "1" is the maximum; it means that we receive i flits per cycle, where i is the number of flits we insert in the FabricPort each cycle.

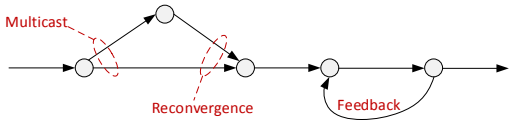


Figure 14: Aspects of complex FPGA applications.

While some of these connections can be mapped onto the NoC, not all of them have to be; the embedded NoC is not meant to be an interconnect capable of connecting everything on the FPGA; rather a flexible low-cost (but high bandwidth) interconnect resource that *augments* the current FPGA traditional interconnect. Remember that the embedded NoC is 1.3% of FPGA core area while the FPGA's traditional interconnect accounts for ~50% [21]. Traditional interconnect can still be used for latency-critical connections while the embedded NoC can be leveraged for connections on which timing closure is difficult or those that require buffering, stallability, or heavy switching.

5. APPLICATION CASE STUDIES

5.1 Simulator

To evaluate the performance of embedded NoCs, we created `RTL2Booksim`[2]: a simulation framework which allows the co-simulation of hardware description languages (HDL) such as Verilog and VHDL, and a widely-used cycle-accurate NoC simulator called Booksim [20].

[2] `RTL2Booksim` is available for download at `www.eecg.utoronto.ca/~mohamed/rtl2booksim.html`

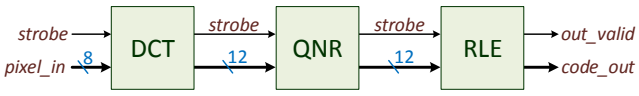


Figure 15: Single-stream JPEG block diagram.

5.2 JPEG Compression

(Latency-sensitive, streaming)

We use a streaming JPEG compression design from [18]. The application consists of three modules as shown in Fig. 15; discrete cosine transform (DCT), quantizer (QNR) and run-length encoding (RLE). The single pipeline shown in Fig. 15 can accept one pixel per cycle and a data strobe that indicates the start of 64 consecutive pixels forming one (8×8) block on which the algorithm operates [18]. 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; the back-end of large social networking websites and search engines. We implemented this parallel JPEG application using direct point-to-point links, then mapped the same design to use the embedded NoC between the modules using **Permapaths** similarly to Fig. 10. 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; however, there are ~8 wasted cycles (equivalent to the zero-load latency of three hops) at the very beginning in the NoC version while the NoC link pipeline is getting populated with valid output data – these 8 cycles are of no consequence.

5.2.1 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. 18 for 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. Note that the DCT module wasn't placed in a partition as it was a very large module and used most of the FPGA's DSP blocks.

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. 16 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 7×5 logic clusters (area of an embedded router); making the placement less regular and on average slightly lengthening the critical path.

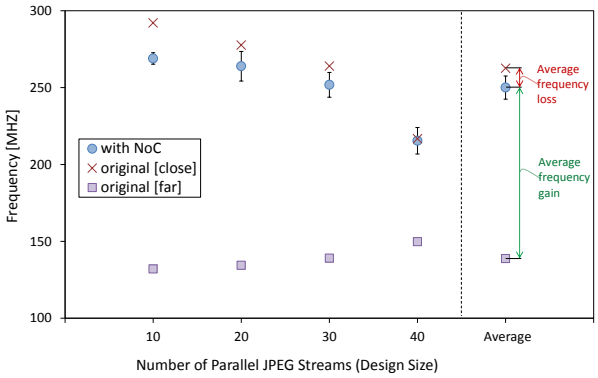


Figure 16: Frequency of the parallel JPEG compression application with and without an NoC. The plot "with NoC" is averaged for the two cases when it's "close" and "far" with the standard deviation plotted as error bars. Results are averaged over 3 seeds.

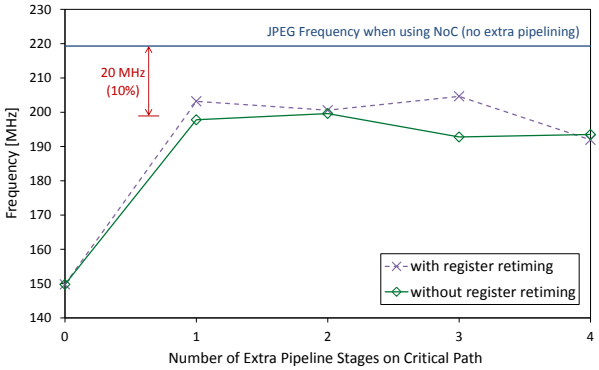


Figure 17: Frequency of parallel JPEG with 40 streams when we add 1-4 pipeline stages on the critical path. Frequency of the same application when connected to the NoC is plotted for comparison. Results are averaged over 3 seeds.

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. 18 – 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. 16. 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 [23]. We plot the frequency of our original JPEG with 40 streams in the "far" configuration after adding 1, 2, 3, and 4 pipeline registers on the critical path, both with and without register retiming optimizations, and we compare to the NoC version frequency in Fig. 17. The plot shows that the frequency of the pipelined version never becomes as good as that of the NoC version even with 4 pipeline stages – the NoC version is 10% better than original JPEG with pipelining.

Table 2: 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) Horizontal (R3,R6)	+13.2% +7.8%	+10.2%
Long	Vertical (C14) Horizontal (R24)	-47.2% -31.6%	-38.6%

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

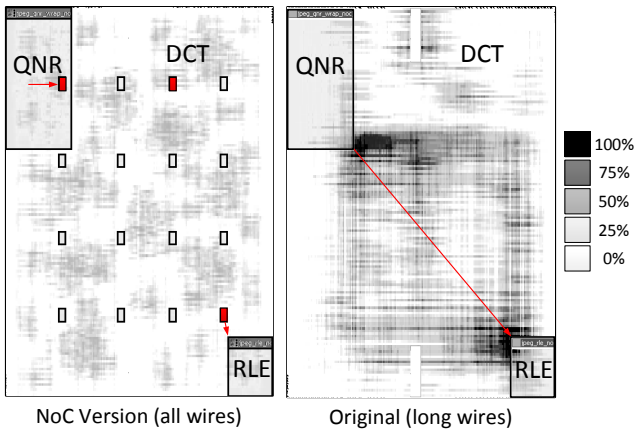


Figure 18: Heat map showing total wire utilization for the NoC version, and only long-wire utilization for the original version of the JPEG application with 40 streams when modules are spaced out in the "far" configuration. In hot spots, utilization of scarce long wires in the original version goes up to 100%, while total wire utilization never exceeds 40% for the NoC version.

5.2.2 *Interconnect Utilization*

Table 2 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.

Fig. 18 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. 18.

5.3 Ethernet Switch *(Latency-insensitive, streaming)*

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 [7]. We describe here how such an Ethernet switch can take full advantage of the embedded NoC, while demonstrating that it considerably outperforms the best previously proposed FPGA switch fabric design [11].

Table 3: Hardware cost breakdown of an NoC-based 10-Gb Ethernet switch on a Stratix V device.

	10GbE MACs	I/O Queues	Translators	**Total**
ALMs	24000	3707	3504	**31211**
M20Ks	0	192	0	**192**

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. 19 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 3. The translator also strips away the NoC control bits before inserting the Ethernet data into the output queue. The design is synthesized on a Stratix V device. A breakdown of its FPGA resource utilization is shown in Table 3. Because we take advantage of the NoC's switching and buffering our switch is ~3× more area efficient than previous FPGA Ethernet switches [11].

Two important performance metrics for Ethernet switch design are bandwidth and latency [13]. 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 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 [7]. While previous work on FPGA switch design has achieved up to 160 Gb/s of aggregate bandwidth [11], 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 [11]). 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. 20 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 [11] 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.

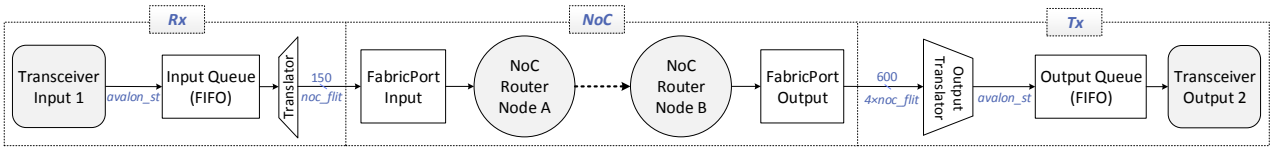


Figure 19: Functional block diagram of one path through our NoC Ethernet switch.

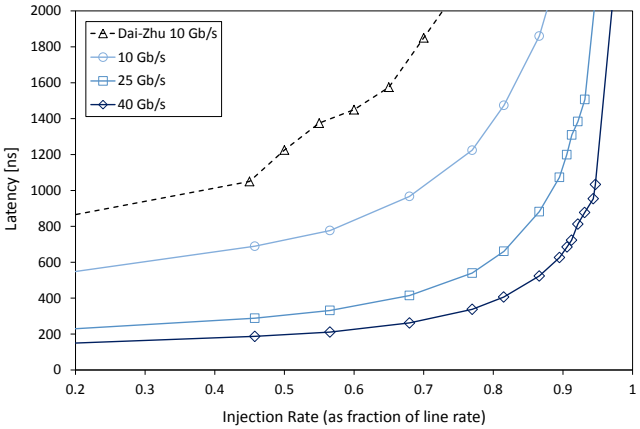


Figure 20: Latency vs. injection rate of the NoC-based Ethernet switch design given line rates of 10, 25, and 40 Gb/s, and compared to the Dai/Zhu 16×16 10 Gb/s FPGA switch fabric design [11]. Our switch queues and Dai/Zhu's switch queues are of size 60kb and 16kb, respectively.

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. We have shown that latency-insensitive systems can be interconnected using an embedded NoC with lower hardware overhead by taking advantage of the NoC's built-in buffering. Additionally, we showed how latency-sensitive systems can be guaranteed fixed delay and throughput through the NoC by using Permapaths.

We investigated two streaming applications; latency-sensitive JPEG that only requires wires between modules, and a latency-insensitive Ethernet switch that requires heavy arbitration and switching between its transceiver modules. With an embedded NoC, JPEG's frequency can be improved by 10–80%. Wire utilization is also improved, as 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. Finally, we 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.

7. ACKNOWLEDGMENTS

We are indebted to Prof. Natalie Enright-Jerger and her research team (Shehab Elsayed, Mario Badr and Robert Hesse) for NoC discussions and for providing some of the code used to build `RTL2Booksim`. We would also like to thank David Lewis, Mike Hutton, Dana How and Desh Singh for feedback on FPGAs, and Kevin Murray for feedback on latency-insensitive design. This work is funded by Altera, NSERC and Vanier CGS.

8. REFERENCES

[1] M. S. Abdelfattah. FPGA NoC Designer. `www.eecg.utoronto.ca/~mohamed/noc_designer.html`.
[2] M. S. Abdelfattah and V. Betz. Design Tradeoffs for Hard and Soft FPGA-based Networks-on-Chip. In *FPT*, pages 95–103, 2012.
[3] M. S. Abdelfattah and V. Betz. The Power of Communication: Energy-Efficient NoCs for FPGAs. In *FPL*, pages 1–8, 2013.
[4] M. S. Abdelfattah and V. Betz. Networks-on-Chip for FPGAs: Hard, Soft or Mixed? *TRETS*, 7(3):20:1–20:22, 2014.
[5] M. S. Abdelfattah and V. Betz. The Case for Embedded Networks-on-Chip on Field-Programmable Gate Arrays. *IEEE Micro*, 34(1):80–89, 2014.
[6] Altera Corp. Video and Image Processing Suite, 2014.
[7] A. Bitar et al. Efficient and programmable Ethernet switching with a NoC-enhanced FPGA. In *ANCS*, 2014.
[8] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, 2002.
[9] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *FPGA*, pages 97–106, 2011.
[10] D. U. Becker. *Efficient Microarchitecture for Network on Chip Routers*. PhD thesis, Stanford University, 2012.
[11] Z. Dai and J. Zhu. Saturating the Transceiver BW: Switch Fabric Design on FPGAs. In *FPGA*, pages 67–75, 2012.
[12] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, Boston, MA, 2004.
[13] I. Elhanany et al. The network processing forum switch fabric benchmark specifications: An overview. *IEEE Network*, 19(2):5–9, 2005.
[14] N. Enright Jerger, L.-S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *ISCA*, pages 229–240, 2008.
[15] R. Francis and S. Moore. Exploring Hard and Soft Networks-on-Chip for FPGAs. In *FPT*, pages 261–264, 2008.
[16] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test*, 22(5), 2005.
[17] K. Goossens et al. Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In *NOCS*, pages 45–54, 2008.
[18] A. Henson and R. Herveille. Video Compression Systems. `www.opencores.org/project,video_systems`, 2008.
[19] R. Ho, K. W. Mai, and M. A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.
[20] N. Jiang et al. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *ISPASS*, pages 86–96, 2013.
[21] D. Lewis et al. Architectural Enhancements in Stratix V. In *FPGA*, pages 147–156, 2013.
[22] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *ICCAD*, pages 227–231, 2003.
[23] K. E. Murray and V. Betz. Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs. In *FPGA*, pages 223–232, 2014.
[24] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.
[25] D. J. Sorin et al. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.
[26] Xilinx Inc. Virtex-5,6,7 Family Overview, 2009-2014.
[27] A. Ye and J. Rose. Using Bus-based Connections to Improve Field-programmable Gate-array Density for Implementing Datapath Circuits. *TVLSI*, 14(5):462–473, 2006.