LYNX: CAD FOR FPGA-BASED NETWORKS-ON-CHIP

Mohamed S. Abdelfattah and Vaughn Betz

Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
{mohamed,vaughn}@eecg.utoronto.ca

ABSTRACT

We present a computer-aided design (CAD) tool that automatically connects an FPGA application using an embedded network-on-chip (NoC). After discussing the CAD flow steps, we delve into the details of implementing transaction communication using our CAD tool. This request-reply type of communication requires special consideration on FPGAs, for example: low round-trip latency, fair arbitration and correct ordering. We show how to implement transaction communication using embedded NoCs, and show that we can improve latency, throughput and efficiency compared to soft buses generated by a commercial CAD tool.

1. INTRODUCTION

Embedded networks-on-chip (NoCs) have been proposed as an addition to large FPGA devices to facilitate interconnecting FPGA applications. It has been shown that embedded NoCs can interconnect wide datapaths more efficiently than multiplexer-based buses built from soft logic [1, 2]. Furthermore, embedded NoCs make design easier by simplifying timing closure, especially to I/O and memory interfaces such as DDRx [3]. Previous work has also focused on making an embedded NoC *usable* by current FPGA applications without a change in design style; for example, streaming communication is implemented properly by enforcing basic ordering constraints and providing some latency guarantees [4].

To use an embedded NoC, an FPGA designer would typically need to know some NoC specifics; the packet format for instance. The designer would also need to decide where (which router) to connect their application module on the NoC. Additionally, the designer will have to sometimes create soft logic wrappers to make NoC communication possible and high performance. This "manual" design, while flexible, is time-consuming, repetitive and possibly suboptimal.

This work explores automating the interconnection of FPGA applications using an Embedded NoC. We introduce **LYNX** : a computer-aided design (CAD) tool that takes an application and an NoC architecture as input, and it connects the given application to the specified NoC architecture – both embedded and soft NoCs are compatible but we focus on the embedded NoCs presented in prior work [4]. **LYNX** automates the *packetization* of data, instantiates any necessary soft logic, selects the routers to which to connect each application module and generates Verilog output for simulation and synthesis.

After presenting the **LYNX** CAD flow in Section 2, we focus on an important communication style used in FPGA applications: transaction communication. Transactions consist of requests and replies, typically between so-called masters and slaves. Previous work has shown that streaming communication can be implemented efficiently using embedded NoCs [4]. How does transaction communication differ? How do we efficiently use an NoC to create transaction connections? How does an embedded NoC compare to soft transaction buses? We answer these questions in Section 3, and we show that embedded NoCs also outperform soft NoCs for transaction systems on FPGAs. By supporting both streaming and transaction communication, embedded NoCs can implement *any* FPGA design that is supported by current FPGA system-integration tools.

Thanks to Gordon Chiu and Jimmy Yeap for answering our questions about Qsys. This work is funded by NSERC, Altera and Vanier CGS.

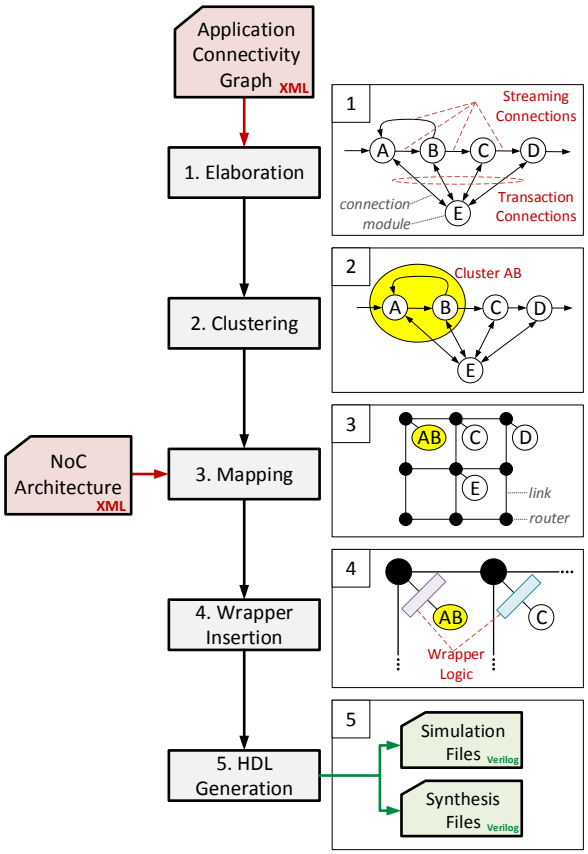


Fig. 1: Overview of the **LYNX** CAD flow for embedded NoCs.

1.1. Related Work and Motivation

There are FPGA system-level interconnection tools in both academia and industry. Xilinx' Vivado IP Integrator [5] and Altera's Qsys [6] can interconnect design modules with standard interfaces (such as AXI or Avalon). These tools

use soft multiplexer-based buses to connect an application. In academia, GENIE [7] is a system integration tool that can generate both fine-grained and system-level interconnect given the description of an application. GENIE uses a soft packet-switched NoC that consists of split and merge blocks for its system interconnection [8]. CONNECT [9] is a web program that generates a soft NoC with custom parameters and topology; however, CONNECT only generates the NoC but does not integrate a full application. Another NoC generator is Hoplite [10]. Hoplite is a very light-weight NoC that uses little resources but is only suitable for applications that can tolerate high-latency and out-of-order data delivery.

In the context of multiprocessors, NoC CAD tools are well-studied [11]; however, we found that mapping FPGA design styles onto an NoC is very different and has its own set of constraints as detailed both in previous [4] and this work. FPGA modules can have any width; multiple modules can share one NoC router, and one module can use multiple NoC routers – a primary difference to the mainly-homogenous multiprocessor systems. FPGA communication protocols – streaming and transactions – also require latency, performance and ordering guarantees that differ from those in multiprocessors. For example, multiprocessors often handles out-of-order data in the processing cores themselves, whereas FPGA modules typically assume that data arrives in-order to avoid data/control hazards, and so FPGA system-level interconnects – such as that generated by Altera's Qsys [6] – guarantee in-order delivery.

A cornerstone of modular hardware design is the decoupling of application modules and the system-level interconnect. Ideally, this interconnect should satisfy the timing requirements of the application, both among the application modules themselves, and in transferring data to fast I/O interfaces such as memory. The importance of a timing-closed, automatically-connected FPGA application is especially evident in high-level synthesis (HLS) [12] and data center acceleration [13]. In both of these important emerging areas, the designer *relies* on an abstraction in which s/he designs the application "kernels", and then an automatic tool connects the application kernels together. In this paper we advocate the use of an embedded NoC to abstract system-level communication, and we present a CAD system (**LYNX**) to automatically leverage NoCs to connect *any* FPGA application. To the best of our knowledge, **LYNX**, is the first FPGA CAD system to target system-level interconnection using hard and soft packet-switched NoCs.

2. LYNX CAD FLOW

LYNX[1] is a CAD tool that automatically connects an FPGA application using an NoC. We connect application modules to NoC routers and generate any soft-logic wrappers that are required for semantically correct and high-performance communication. We start with an annotated application connectivity graph (ACG). In the most basic form, the ACG is simply a definition of the modules in a system and the connections between them, the width of these connections and their type (data, ready or valid). Using only this application metadata, **LYNX** implements the application's connections using the NoC by connecting the application modules to the NoC – the FPGA designer does not need to know anything about how the NoC works to use **LYNX**.

Fig. 1 shows an overview of the **LYNX** CAD flow. An application is entered as a LYNXML[2] file like the example shown in Listing 1. The application is then elaborated and an internal graph representation of the design is created, additionally, connections are labeled either "streaming" or "transaction" as they are treated differently in later stages of the flow. The next step clusters tight latency-critical feedback loops and marks them to be implemented in light-weight low-latency soft connections to avoid throughput degradation [14]. Next, eligible modules or clusters are mapped to available NoC routers. Following mapping, soft-logic wrappers are added, primarily to abstract NoC communication details such as packetization or to manage traffic as we discuss in Section 3. Finally, simulation and synthesis files are generated to be able to use **LYNX** results with traditional synthesis and simulation tools.

```
<!-- Modules -->
<module name="src1">
  <bundle name="obun">
   <port width="128" name="o_y" type="data"/>
   <port width="1" name="o_valid" type="valid"/>
   <port width="1" name="o_ready" type="ready"/>
 </bundle>
</module>

<module name="dst1">
 <bundle name="ibun">
  <port width="128" name="i_x" type="data"/>
  <port width="1" name="i_valid" type="valid"/>
  <port width="1" name="i_ready" type="ready"/>
 </bundle>
</module>

<!-- Connections -->
<connection start="src1.obun" end="dst1.ibun"/>
```

Listing 1: Sample LYNXML description of two modules connected by a 128-bit wide connection.

2.1. Definitions

Before going through each of **LYNX** CAD steps in detail, we define some of the terms we use to avoid ambiguity:

- A **flit** is the smallest unit of data that can be transferred over an NoC.
- A **packet** consists of one-or-more flits, it defines a logical *word* transferred over an NoC.
- A **virtual channel (VC)** refers to a separate buffer that allows the logical separation of data transported over a physical channel such as an NoC link.
- A **bundle** is a collection of ports in an application module, and must have data, valid and ready signals.
- A **connection** (**s**, **d**) exists between a single source bundle (**s**) and a destination bundle (**d**) to which it sends data.
- A **link** is the metal wiring between two routers in an NoC.
- **Streaming communication**: One-way communication between a source and a destination module.
- **Transaction communication**: Two-way communication that consists of a request from a master module and a corresponding reply from a slave module.

[1] **LYNX** is available for download from www.eecg.utoronto.ca/~mohamed/lynx

[2] LYNXML is an XML format that describes an ACG. We hope to standardize this format for system-level interconnect research and evalution.

2.2. Elaboration

In this first step of the **LYNX** CAD flow, we parse the LYNXML description of the design and create an internal graph representation of the system. This graph representation resembles the LYNXML description very closely; it has a design object which contains a list of module objects, that have bundles and ports. Connections are defined as a list of (start,end) bundle pairs in the design object.

The elaboration step takes the design graph as input and classifies the connections into streaming and transaction connections. Elaboration also groups transaction connections to-and-from the same modules together into a "connection group" as shown in the illustration of Fig. 1. This is because the number of modules in a transaction system affects later steps in the CAD flow as we discuss in Section 3.

2.3. Clustering

A major limitation of NoCs is that their latency cannot be reduced beyond the latency required to traverse 1 router – in our embedded NoC, this latency is approximately 8 clock cycles. Consider an application that contains a feedback loop like that illustrated in Fig. 1. The higher the latency of the feedback connection, the lower the throughput. In fact, the throughput of a feedforward system with one feedback connection is equal to $\frac{1}{latency}$ where $latency$ is the latency of the feedback connection.

This is not a new problem, and has been studied extensively in the context of latency-insensitive systems where we choose where to insert pipeline registers that add latency [14]. In this work, the authors use "Tarjan's algorithm" [15] before deciding where to insert pipeline registers to avoid adding any latency to feedback connections. Tarjan's algorithm identifies *strongly-connected components* – defined as nodes of a graph that are connected together in a cycle – and outputs them as a cluster. Similarly, we also use Tarjan's algorithm to cluster modules that are involved in a feedback connection. However, we ignore transaction connections in this clustering step as they inherently contain a feedforward and a feedback connection, and we discuss how to implement these connections efficiently, even using multi-cycle-latency NoC links, in the following Section. On the other hand, feedback streaming connections must be implemented using a low-latency lightweight interconnect to avoid throughput degradation. **LYNX** conservatively clusters streaming modules in a feedback cycle and directly connects ports in this cycle using soft point-to-point links.

2.4. Mapping

Mapping is the core algorithm of the **LYNX** flow – it connects application modules to NoC routers. As shown in Fig. 1, mapping takes an NoC architecture file as input. By changing NoC parameters such as link width, number of routers or routing algorithm, we can optimize a soft NoC for an application. In the case of architecting an embedded NoC before FPGA manufacture, the system architect can use **LYNX** to try out different NoCs for important application benchmarks before deciding on the final architecture.

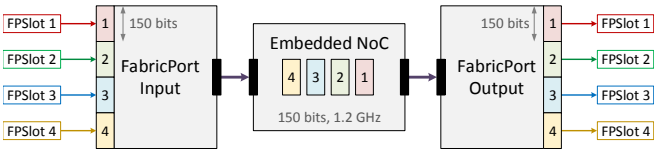


Fig. 2: A FabricPort time-multiplexes wide data from the FPGA fabric to the narrower/faster embedded NoC. 1–4 different bundles can connect to the shown FabricPort by using one-or-more FabricPort Slots (FPSlots) depending on the widthand number of VCs.

Table 1: Possible FabricPort input/output configurations for a time-multiplexing ratio of 4 and 2 VCs. Two modes are unavailable in the FabricPort output because we are limited by 2 VCs.

# Bundles × Width	Input	Output
1 × 4 flits	✓	✓
2 × 2 flits	✓	✓
4 × 1 flit	✓	✗
1 × 2 flits + 2 × 1 flit	✓	✗
1 × 3 flits + 1 × 1 flit	✓	(✓)*

*Possible, but not yet implemented in **LYNX**.

2.4.1. *FabricPort Configurability*

An embedded NoC is typically ~4× faster than the FPGA application. This is why we use a FabricPort to time-multiplex data from an application onto the embedded NoC [4]. Fig. 2 shows how data moves from a FabricPort input, across NoC, then a FabricPort output – a FabricPort exists at each NoC router to perform this width/frequency bridging. The NoC architecture file specifies a time-multiplexing ratio, so any FabricPort (or the absence of one) can be modeled in **LYNX**.

Each FabricPort Slot (FPSlot) is equal to the NoC width (or flit width, which we set to 150 bits), and so each input FPSlot can be used independently by an application bundle. For example, if our bundles are less than 150 bits, we can connect 4 of them to the FabricPort input, each to one FPSlot. In this scenario, each bundle's data will be sent as a 1-flit packet across the NoC. However, a wide 600-bit bundle will use all 4 FPSlots at a router, and it will transfer its data as a 4-flit packet on the NoC.

At the FabricPort output, using the FPSlots independently imposes an additional constraint: each bundle connected at the FabricPort output must receive data on a different VC. Each VC can be stalled separately. This decouples the bundles so that if two bundles are connected at a FabricPort output and one of them stalls (not ready to receive data), the other can continue to receive data on a different VC. This also ensures deadlock freedom [4]. So the maximum number of bundles possible at a FabricPort output is equal to whichever is smaller: the time-multiplexing ratio or the number of VCs. To clarify, table 1 shows the possible FabricPort configurations for an embedded NoC with time-multiplexing ratio of 4, and 2 VCs. The two unavailable configurations at the FabricPort output would require more than 2 VCs to work.

2.4.2. ***LYNX** Mapping*

Mapping is the CAD step that assigns (maps) application modules onto NoC routers – more specifically, mapping assigns bundles to FPSlots. Wide bundles can use one-or-more FPSlots, while multiple narrow bundles can use FPSlots at the

same router, effectively sharing the router. For the mapping to be legal, it only needs to be in agreement with the rules described in Section 2.4.1

LYNX uses simulated annealing to map an application to an NoC. Initially, all bundles are assigned to "off-NoC", and then the high cost of off-NoC bundles quickly forces bundles to connect to NoC routers. The mapping cost function has four components:

- **Path Bandwidth**: To avoid NoC congestion, we add the bandwidth utilization of each NoC link to the cost function if the utilization of a link is greater than 100%. The higher the overutilization of an NoC link, the more it contributes to the cost function. An overutilized NoC link inevitably results in stalling due to contention for resources. Note that the computation of link utilization depends on the packet routing function specified in the NoC architecture file. If overutilized NoC links remain after mapping is complete, warnings are printed out to the screen as this can result in throughput degradation.
- **Latency**: The zero-load latency of each connection is added to the cost function so that we minimize application latency.
- **Multiple-router modules**: If a module has bundles that are connected to more than one router, we penalize the cost function heavily as this mapping may result in highly constrained placement and routing.
- **Off-NoC**: We penalize bundles that are not yet mapped on the NoC depending the number of connections using this bundle. If a bundle has many connection and is left off-NoC, it will require expensive soft logic to connect to the rest of the application. **LYNX** maximizes the use of an embedded NoC – to leverage that hard resource and minimize additional soft interconnect – by prioritizing the mapping of highly connected bundles on the NoC, and giving less-connected bundles lower priority.

Equation 1 is the simulated annealing cost function used in mapping. W_{1-6} are constant weights that control the contribution of each component – they reflect the *importance* of each cost component. $Util$ is a function that returns the link utilization: the total bandwidth of connections that use that link divided by the link bandwidth capacity. $Latency$ is a function that returns the number of cycles of a connection's path on the NoC assuming zero traffic. $OffNoC$ is a boolean function that specifies whether the connection's start/end bundles are mapped on the NoC or not. $Routers$ is a function that returns the number of routers to which this module is connected; ideally, each module should not connect to more than one router.

$$
\begin{aligned}
Cost \;=\; & \sum_{L_i \in Links} W_1 \big(Util(L_i)\big)^{W_2} \\
& + \sum_{C_i \in Conns} \Big(W_3 Latency(C_i) + W_4 OffNoC(C_i) \Big) \\
& + \sum_{M_i \in Modules} W_5 \big(Routers(M_i)\big)^{W_6} \qquad (1)
\end{aligned}
$$

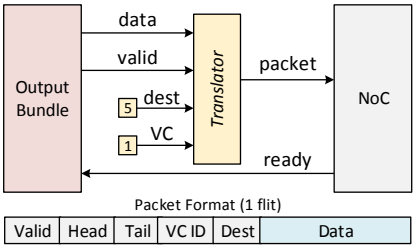


Fig. 3: The simplest translator takes data/valid bits and produces an NoC packet. **LYNX** determines the destination/VC bits statically if the sending output bundle has only one destination as shown; otherwise, the application logic has to set the dest/VC for each outgoing data.

2.5. Wrapper Insertion

"Wrappers" encompass any soft logic required to make communication on the NoC possible and high performance. **LYNX** currently generates three types of soft wrappers: translators, traffic managers and response units. Traffic managers and response units are only required for transaction systems and are discussed in detail in the following section. Translators are required between any bundle and an NoC router port to translate data and control signals into the format of an NoC packet.

Most translators are very simple as they only need to put data and control bits in their correct positions in a packet, and sometimes append more control bits to a packet. For example, a translator automatically appends the destination router address and VC ID if a bundle has only one connection to one destination as shown in Fig. 3. However, if a bundle may send to one-of-many destinations, then the user logic has to specify the destination router and VC, and input them to a translator which will pack those control bits in their correct position.

We currently have four variations of translators to properly interact with streaming/transaction connections, and with different traffic managers. **LYNX** determines which translator to instantiate based on the type of connection and traffic manager, and automatically connects it in the system.

2.6. HDL Generation

The hardware description language (HDL) generation step outputs simulation and synthesis files that can interact with other CAD tools to evaluate the performance and efficiency of **LYNX** NoC interconnect. Embedded NoCs do not currently exist on FPGA devices, or in FPGA vendor tools – how then do we simulate and synthesize designs with an embedded NoC? The simulation output connects the user design to a simulation model of the embedded NoC through **RTL2Booksim**, and generates scripts that simulate the entire system in Modelsim. For synthesis, we use Altera's Quartus II tools. We lock down partitions that have the same size, location and port-width as embedded NoC routers, then we connect the user design to wrappers and to these router "partitions" to accurately measure area and frequency of the design, and to model any physical design artifacts. The same methodology was used in previous work [4].

2.6.1. Mimic Flow: Simulation and Synthesis

In the begining of this paper, we asserted that we only need the ACG to be able to evaluate a candidate interconnect for an application – how can we evaluate an interconnect in absence of the actual application modules that it connects? We

simply instantiate dummy modules in CAD steps that we call "MimicSim" and "MimicSyn". In the simulation scenario, we instantiate dummy traffic generators and analyzers for each output and input bundle respectively. These dummy modules produce a trace file of all the packet transfers which **LYNX** uses to evaluate the throughput and latency at each point of an ACG. This allows us to accurately measure the latency and throughput of an application without having the actual implementation of each application module.

In MimicSyn, we replace the application modules with dummy modules that are heavily pipelined so as not to limit frequency. These dummy modules contain a mix of logic, RAM and arithmetic units. We can tune the ratio of logic/RAM/arithmetic and the overall size of the modules through parameters to better model the actual application being evaluated. By synthesizing an application using dummy modules, we can estimate the area, frequency and power of the application's system-level interconnect, whether it is an NoC or something else.

By using these "mimic" flows, we can evaluate and more-importantly compare system-level interconnect using only the ACG, without the need for the actual applicaion module implementation. This would better allow the fast investigation of different system-level interconnects without a set of complete applications as a prerequisite.

3. TRANSACTION COMMUNICATION

Communication in FPGA applications can be classified into two main types: streaming or transaction (sometimes referred to as "request-reply" or "memory-mapped") communication. Streaming is the simpler of the two, as data only flows in one direction from a source module to a sink module. In transaction communication, a request goes from a master to a slave, and then a reply comes back from the slave to the master. NoC communication is inherently streaming, because data is packetized and sent from one source to one destination. We implement transaction communication on NoCs using two underlying streaming transfers – one for the request, and another for the reply. Additionally, we found that we require careful orchestration of requests/replies using soft wrappers to implement transactions on NoCs efficiently.

In this section, we perform an in-depth treatment of transaction communication, and show how **LYNX** implements transactions using our embedded NoC. Specifically, we discuss how to get reasonable performance and latency of transactions, how to implement fair arbitration, and the role of transaction ordering and its high-performance implementation. Most of the techniques we discuss in this section are not specific to NoCs, and can be used with any system-level interconnect; however, our techniques are particularly effective in multi-cycle-latency interconnects such as NoCs.

To present our results in the context of current systems, we compare the performance and efficiency of transaction systems when implemented using **LYNX** and an embedded NoC, compared to soft buses generated by a commercial system integration tool: Altera Qsys. The embedded NoC we use has a 150-bit link width, 16 nodes, 4 VCs, 10 buffer words per VC and a time-multiplexing factor of 4 – we refer the reader to previous work [3, 4] for further information about the embedded NoC architecture.

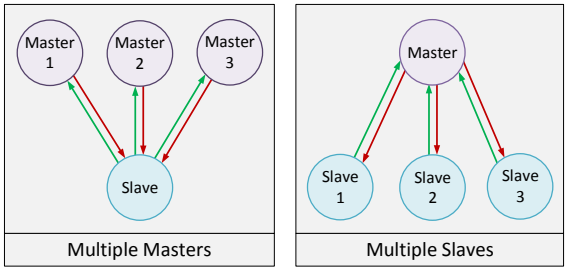


Fig. 4: Transaction systems building blocks.

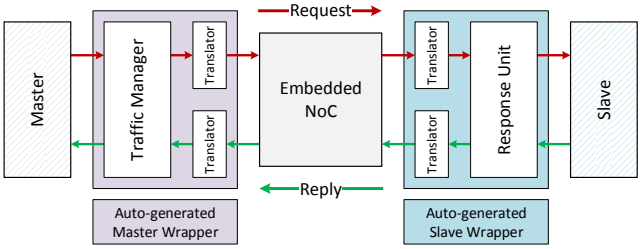


Fig. 5: Master-slave system using the NoC.

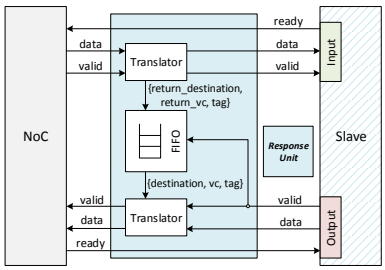


Fig. 6: A response unit at a slave buffers the return address information and attaches it to the slave reply.

In the general case, transaction communication occurs between any number of masters and any number of slaves. However, a multiple-master multiple-slave system can be constructed from its building blocks: multiple-master single-slave, and single-master multiple-slave systems as depicted in Fig. 4. After presenting the transaction system components, we discuss multiple-master and multiple-slave systems in Sections 3.2 and 3.3 respectively. The methods we present with each type of system are composable and can easily be used together in multiple-master multiple-slave systems.

3.1. Transaction System Components in NoCs

Fig. 5 shows how we connect masters and slaves using an NoC. At both the master side and the slave side, **LYNX** automatically generates wrappers to implement transactions. A master makes a request, and the request is only permitted to be issued if a "traffic manager" allows it. This master request then goes through a translator that formats the request data and any control fields into an NoC packet. The request packet then traverses the NoC until it arrives at the slave where it goes through another translator that extracts the data/control fields from the request packet. A response unit then stores some request fields, such as return destination, and later attaches them to the reply issued by the slave.

3.1.1. Response Unit

Fig. 6 shows an implementation for the response unit. A simple translator first inspects the master request and extracts the

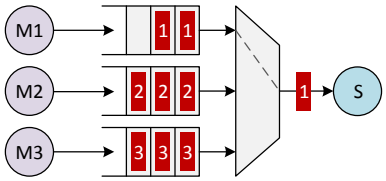


Fig. 7: Requests can build-up quickly in a multiple-master interconnect without traffic management, because all masters can send requests at the same time to a shared slave (which can only process 1 request at a time).

data, valid, return destination, return VC and tag[3]. Data and valid are forwarded to the slave module, while the return information (destination router, VC, tag) is stored in a FIFO. As soon as the slave issues the reply, the return information is automatically attached to the reply using a translator to form a reply packet which can traverse the NoC to the master. Note that the response unit FIFO must be as deep as the number of requests that the slave can handle at any given time so that the FIFO doesn't become full. Also note that this response unit assumes that all replies are issued by the slave in the same order as the requests; otherwise, the slave itself must contain logic to properly tag the replies or reorder them before sending them to the master.

3.2. Multiple-Master Systems

Multiple-master systems are very common in FPGA designs; for example, access to on-chip or off-chip memory; where multiple design modules on the FPGA share memory resources. In such systems it is important to keep latency low and throughput high (to make best use of the shared slave bandwidth). Furthermore, we often need to ensure fair arbitration shares to the masters sharing the same slave. In this subsection, we'll look at systems that have multiple masters and a single slave.

3.2.1. Traffic Build Up (in NoCs)

Before discussing our implementation, we present an important problem that exists in any pipelined multiple-master interconnect. Fig. 7 shows 3 masters connected to 1 slave through FIFO buffers and a multiplexer – this is a simple but valid behavioural model for any multiple-master interconnect; bus or NoC. If every master is constantly sending requests to the slave, request traffic builds up quickly in the interconnect buffering resources because the slave can only process 1 request at a time. Therefore, at steady state, each new request injected into the interconnect effectively waits for every other request that is already buffered, resulting in a high latency equal to $Number\ of\ Masters \times FIFO\ Depth$, where $FIFO\ Depth$ is the number of pipeline stages or buffer locations between a master and the slave.

To keep up with fast I/Os and ever-larger FPGAs, the level of pipelining ($FIFO\ Depth$) in a bus-based interconnect is constantly increasing to ensure that the bus has a high frequency. In NoCs, there are reasonably large buffers (10-flits deep in our case) at each router between a master and a slave, resulting in a very large $FIFO\ Depth$ and a proportionally large latency if traffic builds up in these buffers. This is especially catastrophic for an NoC, being a shared interconnect resource. When the buffers start becoming full, the latency of *all* packets that are using the same routers increases very quickly. To mitigate this problem, we introduce traffic management schemes that avoid traffic build-up altogether.

[3] A tag is an optional field to uniquely identify a request/reply.

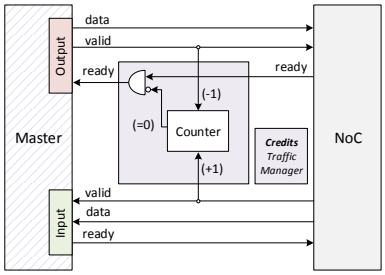


Fig. 8: Credits traffic manager to limit traffic between a master sharing a slave with other masters.

3.2.2. Credit-based Traffic Management

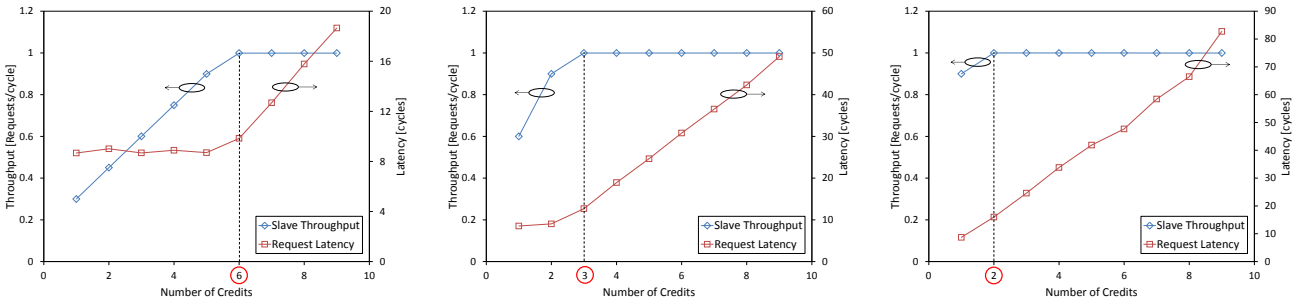
One way to solve the traffic build up problem is to employ a credit-based traffic management scheme. Fig. 8 shows the Credits Traffic Manager, which is placed at each master to limit the number of outstanding requests. The counter in Fig. 8 is set to a selected "number of credits", whenever the master sends a request the credits are decremented by 1, and whenever a reply is received the credits increase by 1. Zero credits stalls the master. This ensures that no new requests are made until replies for the outstanding requests are received.

It is crucial to select the number of credits appropriately – too many credits and traffic will build up, too few credits and that slave will be underutilized. To better visualize this, see Fig. 9: we vary the number of credits for different systems and plot request latency, which we want to keep low, and slave throughput, which we want to be equal to 1. As predicted, increasing the number of credits improves throughput until the slave is fully utilized, then it starts worsening latency beyond that. The ideal number of credits at each master (circled in Fig. 9) depends on the number of masters and the round-trip latency (between sending a request and receiving its reply), and follows the following equation:

$$\text{Credits}_{\text{ideal}} = \frac{\text{Latency}_{\text{roundtrip}}}{\text{Number of Masters}} \quad (2)$$

To understand why equation 2 works, consider several masters communicating with 1 slave. After $\text{Latency}_{\text{roundtrip}}$ cycles, a master receives a reply and therefore increments its credits and is able to send another request. In a 1-master system, the number of credits should be equal to the $\text{Latency}_{\text{roundtrip}}$ so that as soon as the master runs out of credits, a reply arrives and increments the credits by 1 – this ensures that the master is constantly sending requests and the slave bandwidth is fully utilized. Equation 2 effectively shares that slave bandwidth equally by dividing the $\text{Latency}_{\text{roundtrip}}$ by the number of masters.

Fig. 10 plots the ideal number of credits for multiple-master systems while varying the number of masters. The "simulation" data series in this plot was experimentally determined by trying out different number of credits. Equation 2 is also plotted (dotted line) – the model agrees with our simulation results very closely and the discrepencies only exist because our Credits Traffic Managers only support an integer number of credits. We include this model in **LYNX** which



(a) 3 masters. (b) 6 masters. (c) 9 masters.

Fig. 9: Investigation of the ideal number of credits for multiple-master communication with 3, 6 and 9 masters.

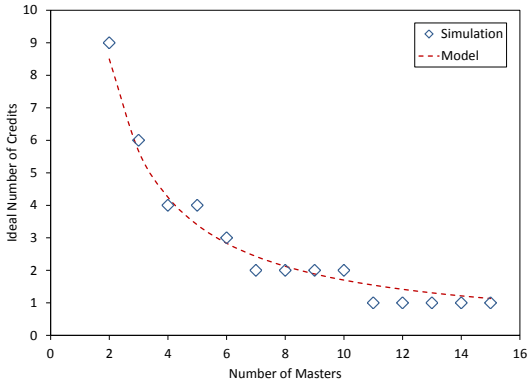


Fig. 10: Ideal number of credits for NoC traffic managers to minimize request latency. Both the experimental evaluation from Fig. 9 is shown, and the model from Equation 2.

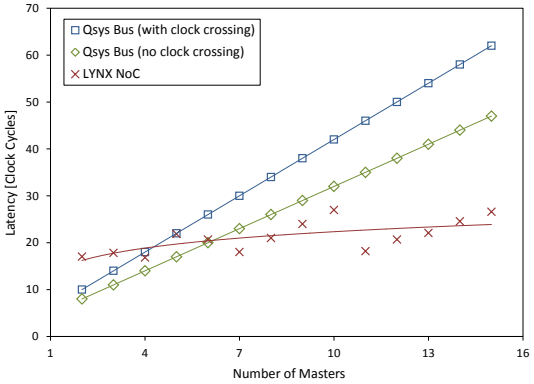


Fig. 11: Comparison of Lynx NoC and Qsys bus latencies in high-throughput systems.

automatically instantiates the traffic manager and sets the correct number of credits for any multiple-master systems.

*3.2.3. Latency Comparison: **LYNX** NoC vs. Qsys Bus*

We generate two pipelined Qsys bus variants for a fair comparison with our embedded NoC. In the one labeled "no clock crossing" in Fig. 11, all masters and slaves operate using the same global clock, whereas "with clock crossing" denotes a system in which all the masters use one clock, and the slave uses a different clock. Qsys generates asynchronous FIFOs to bridge between two clock domains but this adds both area and latency. The embedded NoC contains clock crossing circuitry built-in the FabricPorts so each master and slave can use an independent clock without additional soft logic.

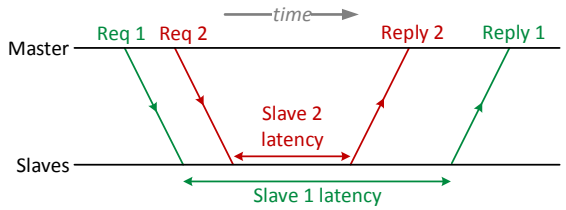


Fig. 12: Requests to multiple slaves can result in out-of-order replies; for example, if slaves have different processing latencies.

Fig. 11 compares the roundtrip latency of multiple-master systems of different sizes. Every master attempts to send a request in each cycle and is stalled by the interconnect (bus or NoC) or a traffic manager. The latency of Qsys buses increases linearly with the number of masters because of traffic build-up as discussed in Section 3.2.1. However, when traffic managers are used in **LYNX**, the latency remains more-or-less constant[4] as we increase the number of masters. Even though the zero-load latency of Qsys buses is close to half that of our embedded NoC (8 cycles compared to 18), proper traffic management in the NoC results in a lower roundtrip latency in a high-throughput multiple-master system.

3.3. Multiple-Slave Systems

An example of a multiple-slave system is a processor (master) controlling multiple slaves such as memory units, accelators or I/O devices. In high-performance FPGA applications, a common multiple-slave system is a banked on-chip or off-chip memory, where a memory is divided into separate banks to provide fast and parallel data storage/access. In this section, we investigate how to build high-performance multiple-slave systems using NoCs while maintaing proper transaction ordering to avoid hazards.

3.3.1. Ordering in Multiple-Slave Systems

Fig. 12 shows the timing diagram of a master connected to two slaves. It sends request 1 to slave 1 first, then request 2 to slave 2, but receives reply 2 before reply 1 because slave 1 has a longer processing latency. Even if all slaves have equal processing latencies, replies can arrive out-of-order because of different interconnect latency or simply if the slave was busy when the request was sent. This out-of-order reply delivery

[4]The fluctuations are due to the difference between the ideal and actual number of credits used. For example, the ideal number of credits for an 11-master system equals 1.54, but we round this value to 1 in our experiments.

can be problematic in an FPGA system where the master expects replies to arrive in order. It is our experience that this is a common assumption in FPGA systems, and that it is left to the interconnect to guarantee correct ordering of transactions. We therefore have to have the option of in-order reply delivery in **LYNX** NoCs to qualify as a system-level interconnect, and have the same correctness properties as existing buses such as those generated by Qsys – we present three ways to do this in the following subsections.

3.3.2. Three Traffic Managers for Multiple-Slave Systems

In this subsection we present three traffic managers to ensure ordering within multiple-slave systems. All three traffic managers include a Credits Traffic Manager (Section 3.2.2) to limit the number of outstanding requests destined to each slave – we have separate counters for each slave in the system. For each traffic manager we can give an equation for its maximum throughput in terms of:

- $N_{req1slave}$: Number of consecutive requests to the same slave.
- Latency: Roundtrip latency between master and slave.
- $N_{VC}, N_{slave}, N_{credits}$: Number of VCs, slaves, credits.

Stall Traffic Manager: A straightforward way to ensure ordering is to conservatively stall the master whenever the destination slave is changed until all outstanding replies are received. This Stall Traffic Manager – also used in Qsys buses – can hurt throughput considerably if the master switches slaves often; however, it is easy to implement and small. Fig. 13 shows our implementation for the Stall Traffic Manager. A simple control unit keeps track of the current destination slave, and if the destination slave changes, the master request is stalled and buffered in a shallow FIFO until all outstanding replies arrive at the master.

$$\text{Throughput}_{Stall} = \frac{N_{req1slave}}{N_{req1slave} + \text{Latency}} \quad (3)$$

VC Traffic Manager: We leverage VCs to avoid stalling every time the destination slave is changed. The VC Traffic Manager assigns a different VC to each slave, then chooses from which VC to read the replies based on the order in which the requests were sent. The VC Traffic Manager in Fig. 13 inspects the request destination, then allocates a VC for it. For the next request, if the destination is the same it uses the same VC, if the destination is different, it is allocated a different VC. While requests are being sent out, the assigned VCs are stored in a FIFO; this tells the traffic manager the next VC it should read from for correct ordering. Note that if we have more slaves than VCs, then the traffic manager stalls until a VC becomes available (all its outstanding replies arrive).

$$\text{Throughput}_{VC} = \begin{cases} N_{VC}\left(\frac{N_{req1slave}}{N_{req1slave}+\text{Latency}}\right) & N_{VC} < N_{slave} \\ 1 & N_{VC} \geq N_{slave} \end{cases} \quad (4)$$

Reorder Buffer (ROB) Traffic Manager: The ROB Traffic manager adds an 8-bit "tag" to each request it sends out, and it only stalls the master when it runs out of credits, similarly to the Credits Traffic Manager (Section 3.2.2). The response unit (Fig. 6) ensures that the tag for each request is attached to the slave reply on the return path to the master. The ROB Traffic Manager then uses that tag to store the incoming reply in a unique location in a hash table, and these replies are read in the correct order in which they were sent. Note that the number of entries in the hash table must be equal to the number of credits so that there are never any collisions (writing replies to the same location in the hash table). This makes this Traffic Manager very tunable; the more credits we have, the better the throughput, but this also comes at the extra area cost of buffering in the hash table.

$$\text{Throughput}_{ROB} = \begin{cases} \frac{N_{credits}}{\text{Latency}} & N_{credits} < \text{Latency} \\ 1 & N_{credits} \geq \text{Latency} \end{cases} \quad (5)$$

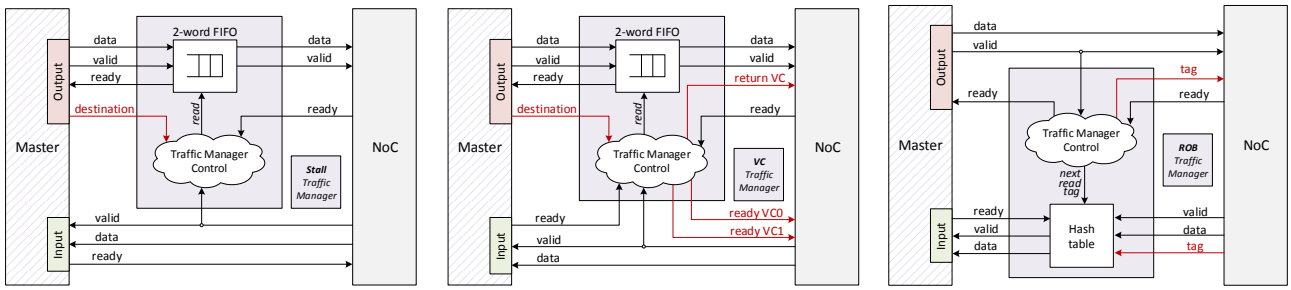
3.3.3. Traffic Managers Performance and Efficiency

Fig. 14 plots the master throughput in multiple-slave systems generated by **LYNX** and Qsys. On the x-axis of Fig. 14, we vary the number of consecutive transfers to each slave. When this value is 1, that means that the master changes the slave it sends to every request – this is the worst-case traffic pattern. In this case, the stall traffic manager performs very poorly as it has to stall each time the slave is changed. It is worse for our embedded NoC compared to Qsys buses because of the higher roundtrip latency. The VC Traffic Manager (with 4 VCs) improves throughput fourfold but must stall because the number of slaves are greater than the number of VCs in Fig. 14; however, if the number of slaves were 4 or less, the throughput would always be the maximum. Finally, an ROB Traffic Manager with $N_{credits} = \text{Latency}$ always has the maximum throughput. With fewer credits, the master throughput decreases linearly until, with 1 credit, the ROB Traffic Manager becomes the same as the Stall Traffic Manager.

Fig. 15 compares the area of the three traffic managers as we vary the number of credits. We measure area in *equivalent* Altera logic clusters (LABs[5]). The Stall and VC Traffic Managers use ~23 LABs, mainly for the 2-word FIFO which is implemented using FPGA flip flops. The ROB Traffic Manager contains sizable hash tables that are implemented as block RAM, and is approximately 1.8× the size of the other traffic managers.

All in all, the traffic managers needed for multislave communication are not large by the standard of today's FPGAs – the smallest Stratix-V FPGA from Altera has 8900 LABs and 688 M20K BRAMS (or 11652 equivalent LABs), and the largest is 46480 equivalent LABs. The largest multiple-master multiple-slave system we can build using our NoC that has a width of 300 bits (for example) will have 30 masters and 2 slaves. In this case we'll need 30 TMs (one per master) for a total area of approximately 690 equivalent LABs – this *absolute* worst case uses between 1.3%–8.0% of the FPGA's LAB + BRAM area, depending on the FPGA size and the selected traffic manager. However, we stress that this pessimistic estimate of additional area overhead is only needed for multiple-slave systems that require a guarantee of ordering, for all other systems we instantiate our Credits Traffic Manager that has a negligible area (less than 1 LAB each).

[5]To compute *equivalent* LABs, we add the logic area (number of LABs) and the block RAM area (Number of BRAMs×4), since each M20K BRAM is as big as 4 LABs [16].



(a) Stall Traffic Manager (b) VC Traffic Manager (c) ROB Traffic Manager

Fig. 13: Different traffic managers to manage communication between a master and multiple slaves.

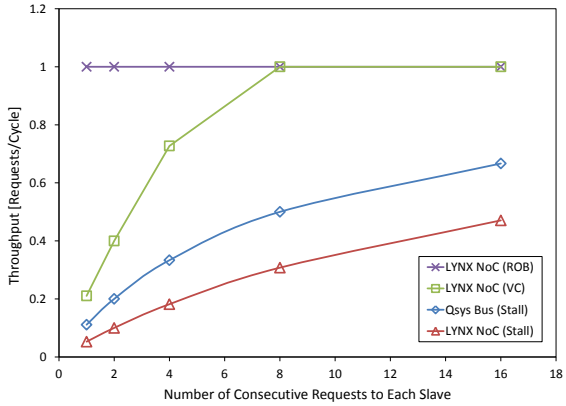


Fig. 14: Maximum master throughput in a multiple-slave system with more than 4 slaves. Number of consecutive requests to the same slave is varied on the x-axis. Different traffic managers are analyzed.

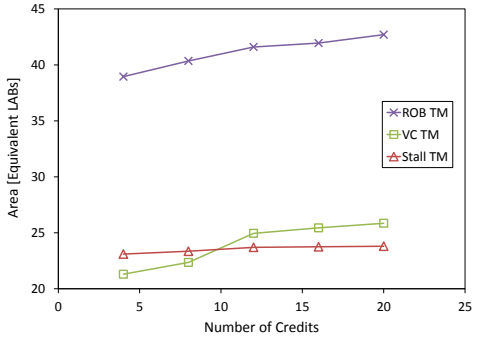


Fig. 15: Area of the different traffic managers with width=300 bits as we increase the maximum number of outstanding requests (credits).

4. AREA & FREQUENCY: LYNX NOC VERSUS QSYS BUS

We have shown how **LYNX** automatically connects an application to an (embedded) NoC. By using traffic managers, embedded NoCs can implement higher-throughput and in many cases lower-latency transaction communication compared to Qsys buses. The **LYNX** CAD system is now comparable to Qsys since it implements most of its features (transactions, streaming, fair arbitration, ordering). In this section, we compare the overall efficiency (area and frequency) of a **LYNX** interconnection solution using an embedded NoC, and a Qsys interconnection solution implemented as a bus.

We use an NoC with 4 VCs, 150-bit width and 16 nodes – this embedded NoC's area is equivalent to 800 LABs [17]. In different FabricPort modes (discussed in Section 2.4.1), we can connect up to 64 modules of 150-bit width, 32 Modules

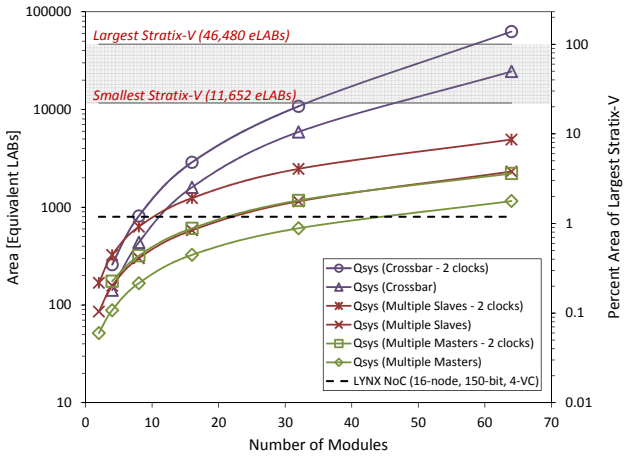


Fig. 16: Area of Qsys buses of varying number of modules and 128-bit width. Different connectivity buses are shown: multiple masters accessing a single slave, single master issuing requests to multiple slaves and a fully connected crossbar of masters and slaves.

of 300-bit width or 16 modules of 600-bit width. We quantify the efficiency of the first option in more detail in this section.

4.1. Area Limit Study

Fig. 16 shows the area of the embedded NoC as compared to any Qsys bus that can implement transaction systems that fit on our embedded NoC. The x-axis shows the total number of modules in a system. For example a 32-module system has 31 masters and 1 slave in a multiple-master system, 1 master and 31 slaves in a multiple-slave system, or 16 masters and 16 slaves in a crossbar system. We also synthesize Qsys buses that have 2 clock domains – a realistic test case for modern FPGAs that can support tens of clocks. In the case of embedded NoCs, clock-crossing circuitry is already included at each router's FabricPort which allows each module to use a different clock [4].

Our 4-VC embedded NoC has an area equivalent to 1.7% of the largest Stratix-V device – this is smaller than most Qsys bus-based systems as shown in the figure. For relatively small buses that interconnect ~10 modules or less, a Qsys soft bus is smaller than the area of an embedded NoC (at best ~8× smaller). However, as the number of modules increase, Qsys bus area increases beyond the area of the embedded NoC. Qsys-generated crossbars are especially huge; a 32x32 crossbar with 2 clock domains is larger than the entire area of the largest Stratix-V FPGA and 78× the area of the embedded NoC. This highlights both the difficulty of imple-

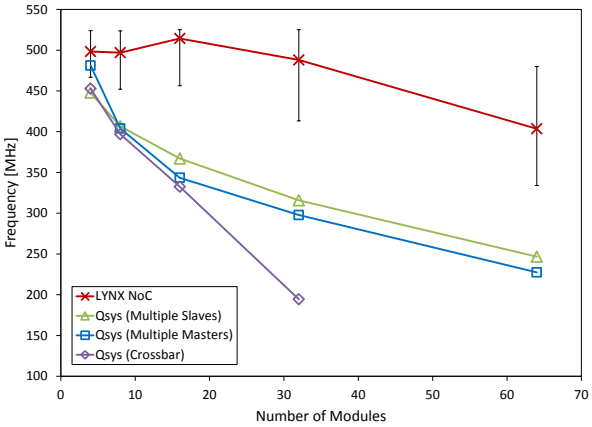


Fig. 17: Comparison of NoC and bus frequency for 128-bit systems. For the embedded NoC, each module can run on an independent clock; the error bars show the range of frequencies for each module.

menting large crossbars on FPGAs[6], and the efficiency of a hard system-level interconnect such as an embedded NoC.

4.2. Frequency Limit Study

Fig. 17 shows the frequency of transaction systems connected by Qsys buses and the **LYNX** embedded NoC. We use the "MimicSyn" flow by connecting dummy modules to the bus and embedded NoC, and measure the resulting overall frequency. The mimic module has a high frequency in isolation (~525 MHz) and it consist of a heavily pipelined array of soft logic (199 LABs), multiplers (7 DSP blocks) and BRAM (15 M20K BRAMs) – this mix of FPGA resources is meant to model an average-case realistic application module that does not limit overall application frequency.

With the embedded NoC, each module in the system operates at an independent clock – we plot the minimum, maximum and average of these module frequencies using error bars in Fig. 17 to show the variation. In highly-connected systems, the minimum frequency will typically govern overall system performance because data will have to be processed by the slowest module – this is true for an application consisting of a cascade of streaming modules. However, in more decoupled systems where there are multiple independent modules processing data in parallel, the average speed of the modules affects performance. For example, if there is a master requesting data from two slave memory modules equally, and these slaves are running at 100 MHz, and 150 MHz, their *effective* frequency is the average (125 MHz) because half the requests will complete more quickly at 150 MHz, while others will run at the slower 100 MHz clock.

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 10x5=50 logic clusters; each one of those partitions represents an embedded hard NoC router with its FabricPorts and interface to FPGA [4]. Fig. 17 shows that, compared to single-clock Qsys buses, the **LYNX** embedded NoC achieves ~1.5× higher frequency on average. Furthermore, the connection pattern does not influence frequency in the embedded NoC as it does for a Qsys bus – the NoC itself does not change, we are just using it in a different way; however, with Qsys buses, the generated bus is different depending on the number of masters and slaves. In the extreme case, a 16x16 Qsys crossbar is 3× slower than our embedded NoC.

[6]More efficient large crossbars can push FPGAs into new markets such as building a high-bandwidth Ethernet switch [4, 18], or implementing hardware mapreduce [19].

5. CONCLUSION

We presented a CAD flow to connect an FPGA application using an NoC. We then focused on the implementation of transaction communication using an embedded NoC; specifically, how to maximize throughput and minimize latency, how to implement fair arbitration in distributed interconnects like NoCs, and how to ensure correct transaction ordering. Our embedded NoC results outperform Altera's Qsys soft customized buses in latency, throughput and area efficiency, especially for larger and higher-throughput systems.

REFERENCES

[1] M. S. Abdelfattah and V. Betz, "Networks-on-Chip for FPGAs: Hard, Soft or Mixed?" *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 20:1–20:22, 2014.

[2] ——, "Power Analysis of Embedded NoCs on FPGAs and Comparison With Custom Buses," *Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 24, no. 1, pp. 165–177, 2015.

[3] ——, "The Case for Embedded Networks on Chip on FPGAs," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, 2014.

[4] M. S. Abdelfattah, *et al.*, "Take the Highway: Design for Embedded NoCs on FPGAs," *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2015, pp. 98–107.

[5] Vivado IP Integrator. [Online]. www.xilinx.com

[6] Altera Qsys. [Online]. www.altera.com

[7] A. Rodionov, *et al.*, "Fine-Grained Interconnect Synthesis," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 46–55.

[8] N. Kapre, *et al.*, "Packet switched vs. time multiplexed fpga overlay networks," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2006, pp. 205–216.

[9] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 37–46.

[10] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," *International Conference on Field-Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[11] P. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60 – 76, 2013.

[12] B. Fort, *et al.*, "Automating the Design of Embedded Systems with LegUp High-Level Synthesis," *International Conference on Embedded and Ubiquitous Computing (EUC)*, 2014, pp. 120–129.

[13] A. Putnam, *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 13–24.

[14] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency Insensitive Systems," *Design Automation Conference (DAC)*. ACM, 2000, pp. 361–367.

[15] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[16] R. Rashid, *et al.*, "Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls," *International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 20–27.

[17] M. Abdelfattah, *et al.*, "Design and simulation tools for Embedded NOCs on FPGAs," *International Conference on Field-Programmable Logic and Applications (FPL)*, 2015.

[18] A. Bitar, *et al.*, "Efficient and Programmable Ethernet Switching with a NoC-enhanced FPGA," *Symposium on Architectures for Networking and Communication Systems (ANCS)*. ACM, 2014, pp. 89–100.

[19] Y. Shan, *et al.*, "FPMR: MapReduce framework on FPGA," *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2010, pp. 93–102.