Embedded Networks on Chip for Field-Programmable Gate Arrays

by

Mohamed Saied Abdelfattah

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Mohamed Saied Abdelfattah

Abstract

Embedded Networks on Chip for Field-Programmable Gate Arrays

Mohamed Saied Abdelfattah
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2016

Modern field-programmable gate arrays (FPGAs) have a large capacity and a myriad of embedded blocks for computation, memory and I/O interfacing. This allows the implementation of ever-larger applications; however, the increase in application size comes with an inevitable increase in complexity, making it a challenge to implement on-chip communication. Today, it is a designer's burden to create a customized communication circuit to interconnect an application, using the fine-grained FPGA fabric that has single-bit control over every wire segment and logic cell. Instead, we propose embedding a network-on-chip (NoC) to implement system-level communication on FPGAs. A prefabricated NoC improves communication efficiency, eases timing closure, and abstracts system-level communication on FPGAs, separating an application's behaviour and communication which makes the design of complex FPGA applications easier and faster. This thesis presents a complete embedded NoC solution, including the NoC architecture and interface, rules to guide its use with FPGA design styles, application case studies to showcase its advantages, and a computer-aided design (CAD) system to automatically interconnect applications using an embedded NoC.

We compare NoC components when implemented hard versus soft, then build on this component-level analysis to architect embedded NoCs and integrate them into the FPGA fabric; these NoCs are on average 20–23× smaller and 5–6× faster than soft NoCs. We design custom interfaces between the embedded NoC and the FPGA fabric to transport data efficiently without compromising on the FPGA's configurability, and then we enumerate the necessary conditions to implement FPGA-compatible communication styles using our NoC. Next, our application case study with image compression shows that an embedded NoC improves frequency by 10–80% and reduces the utilization of scarce long wires by 40%. Additionally, we leverage our embedded NoC to create an Ethernet switch that has ~5× more bandwidth and ~3× lower area compared to other FPGA-based switches. Finally, we create a CAD system (`LYNX`) that automatically connects an application using an embedded NoC. We compare our `LYNX` + embedded NoC interconnection solution to a commercial CAD tool that generates a custom soft bus, and show that we improve both the efficiency and performance of most systems.

Acknowledgements

My sincerest thanks go to my advisor Prof. Vaughn Betz from whom I have learned so much over the past 5 years, both technically, pedagogically and personally. Never have I met someone so knowledgeable, yet so modest and generous with his time and supervision; I will forever be indebted to him. I consider myself lucky to have worked with one of the world's foremost authorities on FPGA technology – his guidance has greatly improved the quality of this work.

I would like to thank Prof. Natalie Enright Jerger for her constant guidance on everything related to NoCs throughout my PhD – her feedback has been invaluable and I am deeply appreciative of our meetings and email exchanges. I was also fortunate to meet many outstanding researchers, both at the University of Toronto and at Altera Corporation, with whom I have regularly consulted and learned from. Thanks to Prof. Jason Anderson, Prof. Jonathan Rose, Prof. Paul Chow, Dr. Desh Singh, Dr. David Lewis, Dr. Mike Hutton and Dr. Dana How. Their expertise and feedback helped make this work more realistic and accurate. I would also like to thank Prof. James Hoe of Carnegie Mellon University for serving as external examiner during my final PhD defense, and for providing fresh insights and valuable comments on my work.

Special thanks to master's graduate Andrew Bitar who joined my project and proved its viability through essential networking application case studies. Andrew's energy accelerated our collaborative work, and contributed to some of the best parts of this thesis. Thanks to my lab mates and friends who were always patient in hearing me talk about my research, and often sparked clever additions and enhancements to my research. Kevin Murray, Jeff Cassidy, Charles Chiasson, Tim Liu, Shehab Yomn and Mario Badr. I would also like to thank summer students Ange Yaghi, Harshita Huria and Aya ElSayed for contributing excellent work my PhD project. I learned a lot by supervising you.

During my PhD, I have been fortunate to receive funding from Altera Corporation, the University of Toronto, the Connaught International Scholarship and the Vanier Canada Graduate Scholarship.

On a more personal note, I would like to thank my friends whose camaraderie made five years of PhD research a lot easier and much more fun. My warmest thanks go to my wife, Lina, a brilliant art historian who was never too bored or impatient when I rambled on and on about FPGAs and NoCs, on the contrary, she was always keen on hearing about my latest work details. Also, her opinions about creating good illustrations greatly improved the quality of the figures and graphs in this thesis. Without your care and love, nothing would be the same. Finally, I would like to thank my parents and siblings for their unconditional love and support which fueled my journey until I got here.

Preface

Work in this thesis is largely based on the following publications:

Peer-reviewed Conference Papers:

- Mohamed S Abdelfattah and Vaughn Betz. Design Tradeoffs for Hard and Soft FPGA-based Networks-on-Chip. In *International Conference on Field-Programmable Technology (FPT)*, pages 95–103. IEEE, 2012
- Mohamed S Abdelfattah and Vaughn Betz. The Power of Communication: Energy-Efficient NoCs for FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013 (Stamatis Vassiliadis Best Paper Award)
- Mohamed S. Abdelfattah and Vaughn Betz. Augmenting FPGAs with Embedded Networks-on-Chip. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, 2013
- Mohamed S. Abdelfattah, Andrew Bitar, and Vaughn Betz. Take the Highway: Design for Embedded NoCs on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 98–107. ACM, 2015 (Best Paper Award)
- M.S. Abdelfattah, A. Bitar, A. Yaghi, and V. Betz. Design and simulation tools for Embedded NOCs on FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2015. [Demonstration Abstract]
- Andrew Bitar, Mohamed S. Abdelfattah, and Vaughn Betz. Bringing Programmability to the Data Plane: Packet Processing with a NoC-Enhanced FPGA. In *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2015
- Mohamed S Abdelfattah and Vaughn Betz. LYNX: CAD for Embedded NoCs on FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2016 (Accepted)

Peer-reviewed Journal Papers:

- Mohamed S Abdelfattah and Vaughn Betz. The Case for Embedded Networks on Chip on FPGAs. *IEEE Micro*, 34(1):80–89, 2014
- Mohamed S Abdelfattah and Vaughn Betz. Networks-on-Chip for FPGAs: Hard, Soft or Mixed? *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–22, 2014 (Invited)

- Mohamed S Abdelfattah and Vaughn Betz. Power Analysis of Embedded NoCs on FPGAs and Comparison With Custom Buses. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 24(1):165–177, 2016
- Mohamed S Abdelfattah, Andrew Bitar, and Vaughn Betz. Design and Applications for Embedded Networks-on-Chip on Field-Programmable Gate-Arrays. *IEEE Transactions on Computers (TCOMP)*, 2016 (Submitted)

Book Chapter and Patent Application:

- Mohamed S Abdelfattah and Vaughn Betz. Embedded Networks-on-Chip for FPGAs. In Pierre-Emmanuel Gaillardon, editor, *Reconfigurable Logic: Architecture, Tools and Applications*, chapter 6, pages 149–184. CRC Press, 2016
- Mohamed S Abdelfattah and Vaughn Betz. Field Programmable Gate-Array with Network-on-Chip Hardware and Design Flow, 04 2015. US Patent Application 14/060,253

Contents

List of Tables

List of Figures

List of Abbreviations

ACG application connectivity graph

aFIFO asynchronous first-in first-out memory

ALM adaptive logic module

ASIC application-specific integrated circuit

BRAM block random access memory

CAD computer-aided Design

DCT discrete cosine transform

DDR3 double data rate version 3

DDRx double data rate version x

DFI ddr-phy interface

DPI direct programming interface

DSP digital signal processing unit

ECC error correcting codes

FIFO first-in first-out memory

FPGA field-programmable gate array

GPU graphics processing unit

HDL hardware description language

HLS high-level synthesis

HOL head of line

I/O input/output

IP intellectual property

LAB logic array block

LUT lookup table

LUTRAM lookup table random access memory

MPFE multi-port front end

NoC network on chip

OS operating system

PHY physical interface

RAM random access memory

RCF routing constraints file

RLE run-length encoding

RTL register-transfer level

SAMQ statically allocated multi-queue

TDM time-division multiplexing

TSMC Taiwan Semiconductor Manufacturing Company

VC virtual channel

VCD value change dump

VLSI very large-scale system integration

XML extensible markup language

Chapter 1

Introduction

Contents

1.1 Motivation

Field-programmable gate arrays (FPGAs) have seen an incredible advancement in their technology and applications over the past two decades. FPGAs were once *small* devices with hundreds of configurable logic elements, wires and multiplexers for interconnection, and simple input/output (I/O) buffers. At that time, FPGAs were mainly used to create small logic circuits to augment application-specific integrated circuit (ASIC) chips. Two decades later, thanks to semiconductor scaling, FPGAs now contain millions of logic elements and are used in a myriad of computing and communication applications. Additionally, the FPGA architecture itself is now significantly more complex. Instead of dominating the chip, logic elements now constitute only one third of the FPGA die area [33]. The other two thirds of the FPGA include embedded block random access memory (BRAM), (floating-point) digital signal processing units (DSPs), multi-core processors and fast I/O controllers to connect to external memory, Ethernet and PCIe. Figure 1.1 illustrates the development of FPGAs.

Surprisingly, the FPGA interconnect is much-less changed over the past twenty years; it still consists of different-length wires that are stitched together using statically-controlled multiplexers to form a connection between two parts of the FPGA. Even though the FPGA's interconnect scaled relatively well, it has started to show signs of stress. Fundamentally, the delay of long wires increases with semiconductor scaling so it is no surprise that the FPGA's interconnect now dominates critical path delay. This problem manifests itself clearly in connecting to fast external memory interfaces – it is becoming increasingly difficult to close timing on soft buses that connect to memory interfaces, thus necessitating multiple time-consuming design iterations. I/O and external memory speeds have been increasing at a faster rate than the FPGA's speed, so the soft buses used to distribute I/O data have to run at a slower speed and are therefore very wide, and consume much area and power.

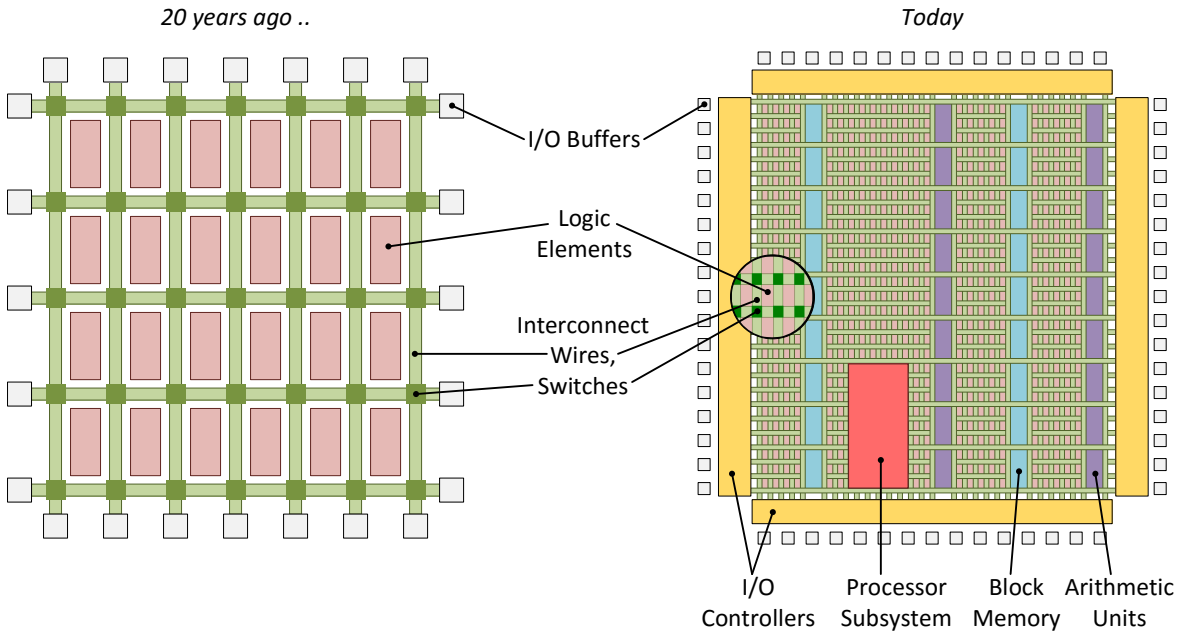


Figure 1.1: Block diagram of FPGAs twenty years ago and today. Modern FPGAs contain many hard blocks to efficiently implement on-chip memory and arithmetic and to connect to I/O interfaces.

The current FPGA interconnect is configurable at a very low level; the user can specify exactly how each wire is used. This fine-grained control over the interconnect is important for creating the small and highly customized connections that are found within a single intellectual property (IP) module. However, modern FPGA applications now include multiple IP modules communicating through standard wide interfaces. In implementing these wide inter-module connections, the low-level configurability of the current FPGA interconnect becomes a burden. For example, consider two modules with a frequency of 400 MHz communicating through a simple 512-bit point-to-point connection. Using the traditional FPGA interconnect, each bit of this connection will be created using a combination of wire segments and multiplexers in the FPGA's interconnect fabric – if 511 out of the 512 bits successfully operated at 400 MHz and only 1 bit failed to meet the timing constraint and ran at 350 MHz, the entire system will be governed by that lower speed. This brittleness in the performance of soft interconnect is common in FPGA applications, especially when connecting to high-bandwidth I/O interfaces, begging the question of whether there is a better way to implement system-level communication on FPGAs.

In addition to the inefficiency of soft buses and their time-consuming design, we believe that they are also a barrier to modular design, which is crucial for modern FPGAs. The physical implementation of a system interconnected with a soft bus is monolithic – it is difficult to independently optimize and compile the modules connected through a soft bus because timing paths extend beyond the modules and into the bus. This dependence forces most designs to be compiled in a single time-consuming compilation and makes it difficult to compile modules in parallel. Furthermore, the performance and size of a soft bus is hard to predict until compilation completes making for a long compile-debug-recompile cycle. Additionally, partial reconfiguration is challenging to implement without clear interfaces to which a newly-configured module can connect to without interrupting a running FPGA application. Finally, any soft bus is specific to the application and FPGA device for which it was designed and therefore not easily

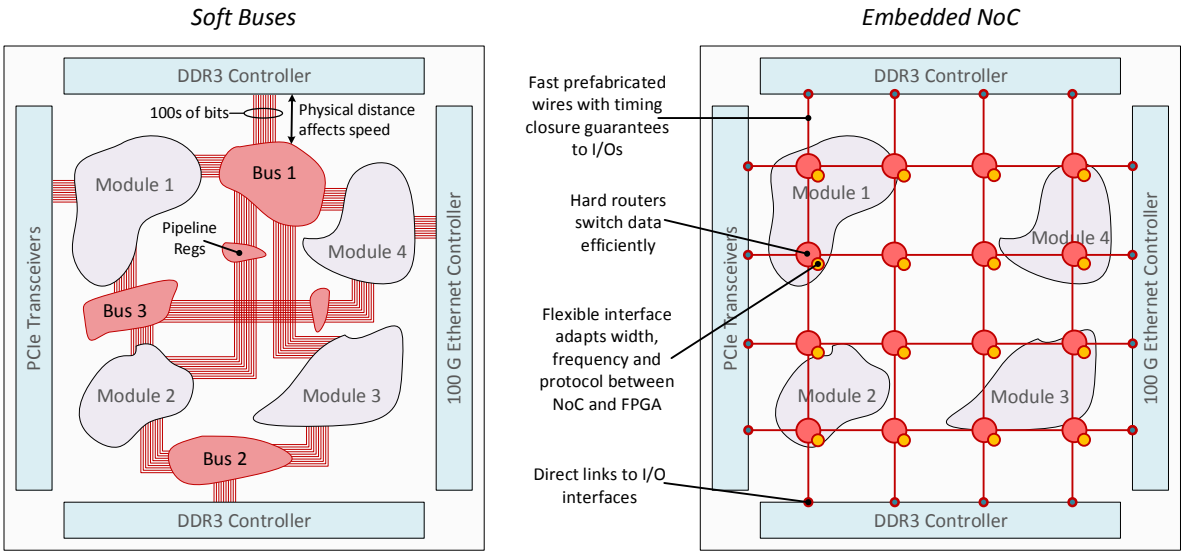


Figure 1.2: System-level interconnection with soft buses or an embedded NoC.

portable to other applications or devices. Motivated by the shortcomings of soft buses for system-level interconnection, we propose a new embedded system-level interconnect for FPGAs.

1.2 Embedded NoCs for Future FPGAs

This thesis proposes augmenting FPGA interconnect with an embedded network on chip (NoC) to implement system-level interconnection. This is similar to current trends in multiprocessor and ASIC chip design where the relative increase in wire delay has pushed designs towards communication-based systems using NoCs. A prefabricated hard NoC will have a predictable speed and will connect directly to I/O interfaces, thus easing much of the timing closure iterations faced by designers of soft buses. Because the NoC is implemented in hard logic, it is typically more area and power efficient than soft bus-based interconnects, especially for high-bandwidth applications. By raising the abstraction of system-level interconnection, an NoC promotes the modular design of FPGA systems and clearly separates computation and communication in an FPGA application. This will enable avenues for the independent optimization and compilation of modules connected through the NoC, and better-suits the increasing complexity of FPGA applications. An FPGA designer needs only to focus on the design of IP modules, and then uses the embedded NoC for interconnection (optionally through an automated computer-aided Design (CAD) flow), which eases design and improves application portability when ported to a different FPGA.

Every sizable FPGA application needs some form of system-level interconnect to transport data among IP modules in the FPGA core and I/O. Therefore, it is worthwhile to embed a flexible system-level interconnect if it suits FPGA design styles and works with important FPGA applications. We believe an NoC can fulfill that purpose – Figure 1.2 compares system-level interconnection with soft buses or the proposed embedded NoC. The proposed NoC consists of hard routers and links to leverage the efficiency and performance gains of hardening. NoC links transport the data between routers, which arbitrate and switch the data to any region on the FPGA. Importantly, the NoC is designed to satisfy

the bandwidth requirements of high-speed I/O and memory interfaces so that it can always transport data from these fast interfaces without the need for manual timing closure. Another key feature of the NoC is its interface to the FPGA fabric. This interface must be flexible in supporting different widths and frequencies, as well as different communication styles and protocols.

1.3 Thesis Organization

After presenting motivating and prior work in Chapter 2, this thesis is divided into three parts. Part I is an efficiency study of NoC implementation on FPGAs. We use the results from our NoC component-level analysis in Chapter 5 to build area, delay and power models for complete NoCs in Chapter 6. In Chapter 7, we prototype an embedded NoC that suits modern FPGAs.

Part II investigates how to use an embedded NoC with FPGA designs. We design a flexible interface between the NoC routers and the FPGA fabric in Chapter 8. We also investigate the implications of connecting an embedded NoC directly to external I/O interfaces in the same chapter. Chapter 9 explains the design rules one must follow to implement different FPGA design styles on NoCs. To be able to test our embedded NoC, we develop simulation and prototyping tools in Chapter 10. Chapter 11 presents three application case studies that highlight the merits of embedded NoCs compared to soft buses.

In Part III, we design a CAD system to automatically connect an FPGA application using an embedded NoC. We discuss the CAD flow in Chapter 12, and then focus on properly implementing transaction communication using our embedded NoC. In Chapter 13, we compare the efficiency and performance of our CAD system to those of Altera's Qsys in implementing transaction systems. Finally, we conclude the thesis in Chapter 14 and enumerate future avenues for research building on our work.

Chapter 2

Background

Contents

On-chip communication is becoming ever-more challenging as very large-scale system integration (VLSI) systems grow larger and more complex; wires alone are no longer a suitable interconnect. We summarize the problems facing modern VLSI systems, and the progression of resulting solutions. This has often led to the use of NoCs to handle communication in several types of chips. We focus on FPGA-specific challenges and solutions, and motivate the need for a new addition to the FPGA's interconnect architecture – we propose an embedded NoC. Finally, we survey existing FPGA-based NoC work.

2.1 Interconnection Problems and Solutions

In this section, we investigate the problem of wire scaling in advanced semiconductor process nodes and how it led to a paradigm shift in how VLSI circuits are designed. First, wire delay was taken into account by making existing CAD flows interconnect-aware. Next, latency-insensitive design decoupled communication and computation to simplify the design of large and complex systems. That led to communication-based system-level design in which on-chip communication is implement by a dedicated and decoupled interconnection circuit such as an NoC. We end this section with a brief overview of work in the domain of NoCs.

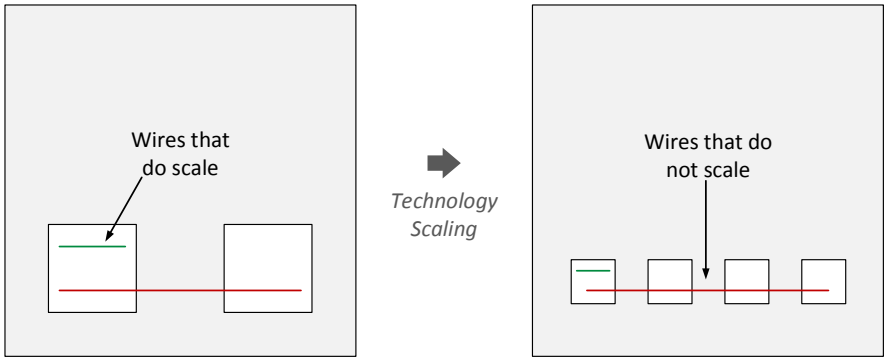
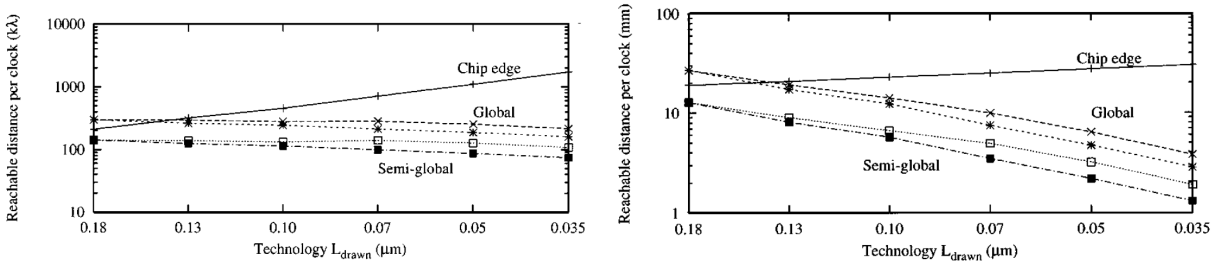


Figure 2.1: Local intra-module wires scale across process technology nodes, but long inter-module wires do not (adapted from [75]).

2.1.1 Scaling of Wires

The number of transistors on a semiconductor chip doubles with every new process technology [110]. As transistors shrink in size, they become faster and generally consume less power [1]. This trend has been the driving force behind the exponential increase in computing resources on a single computer chip for all types of VLSI circuits including FPGAs. However, to make useful computations, we need to connect different parts of the chip together using metal wires. How do these metal wires scale, compared to the exponential scaling of transistors?

Many have predicted that wires will quickly limit the performance of VLSI circuits [32, 51]. In their 2001 paper "The Future of Wires", Ho et al. perform a detailed analysis of this problem [75]. They found wires scale well across process nodes – almost proportionally to transistors – when they span the same *logical* distance; that is, they connect the same number of transistors. This implies that wiring that is local to a specific design unit or module (as shown in Figure 2.1) will be able to keep up with the transistor speed if that design unit is implemented in a newer technology. However, if a wire were to span a specific *physical* distance, say 1 cm for example, it will not be able to keep up with transistor speed at a smaller process technology.



(a) Logical distance.

(b) Physical distance.

Figure 2.2: Reachable distance (logical and physical) per clock cycle using global and semi-global wires across different process technologies (from [75]).

Figure 2.2 summarizes the results succinctly. It shows that wires can traverse roughly the same logical distance in one clock cycle as technology scaling shrinks the transistor size. However, the physical distance reachable per clock cycle decreases drastically with each process node. Ho el al.'s findings were

somewhat surprising compared to conventional wisdom at the time. They conclude that *"...the real problem is not with the wire, but rather with the increasing complexity that scaling enables."*

As transistors become smaller, and their numbers increase exponentially, it becomes difficult to handle the ever-increasing complexity. Wire scaling is just one manifestation of the problem. The problem does not lie within the wires themselves, rather, it is in the way our designs use the wires. It is no longer possible to cross a modern semiconductor chip in one clock cycle using wires [75]. This is why we need to find better ways to cross longer logical distances on a chip; that is, we need to implement more scalable forms of interconnection for our increasingly complex designs. We will refer to this scalable chip-wide interconnection as "system-level interconnect".

2.1.2 Interconnect-Aware Design

For many years, wires were an afterthought in VLSI design: transistors were the key resource. Only at the later stages of design, during layout, were wires planned and optimized. However, poor wire scaling led to wires becoming a major (and eventually dominant) source of delay and sometimes area. This motivated a change in the VLSI design flow where wire delay was to be accounted for early in the design flow. Electronic design automation companies such as Synopsis and Cadence hurried to implement so-called "physical synthesis" in their CAD flows. Physical synthesis accounted for interconnect delay by combining timing analysis and estimates of interconnect delay with synthesis early in the design process, thus allowing early optimizations to aid timing closure [100].

Cong developed an "interconnect-centric" design flow that explicitly accounts for wire delays early in the design process [48]. Instead of waiting until layout to plan wires, Cong proposed starting the design flow with "interconnect topology planning" [48]. This first step translates a design description into a *physical* design hierarchy that attempts to optimize delay primarily based on global wire delay. This is combined with partitioning and register retiming to optimize and estimate delay of the whole design, at which point feedback can be given to the designer on whether they can meet their performance targets. The designer can then change his design and try interconnect topology planning again until his/her timing requirements are met. When that happens, design modules are synthesized and placed within the generated physical hierarchy. Finally, the "interconnect synthesis" step runs before layout to explicitly optimize wire sizes and buffer insertion [48].

Cong's work fell short of addressing all of the concerns that arise with increasing design complexity. His main focus was on optimizing wire parameters to minimize delay and improve signal integrity and he succeeded in modifying the current VLSI CAD flow for that focus. However, when timing requirements were not met, Cong's framework simply gives early feedback to the user to manually change the design architecture and try again [48] – this manual intervention has obvious productivity and efficiency drawbacks.

2.1.3 Latency-Insensitive Design

Around the same time period (late 90s), other researchers were already developing design flows that automate the insertion of pipeline registers on timing-critical paths – this "latency-insensitive design" methodology required both a change in the CAD tools and the design specification, but promised to solve the problem of wire delay. Researchers were moving towards communication-centric design flows in

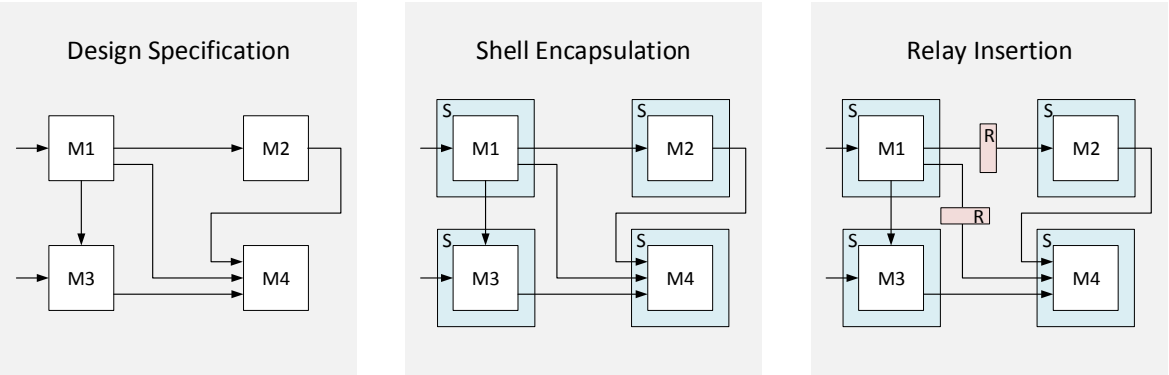


Figure 2.3: Latency-insensitive design flow steps.

which a system-level interconnect was independently designed and used at the heart of a module-based design flow.

Timing Closure Problem

Timing closure means meeting the clock-cycle time constraints of a hardware circuit. The timing closure *problem* arises when we cannot meet those timing constraints, and it is significantly worsened by the wire delay scaling problem. Previously, we could neglect wire delay altogether, but now, wire delay dominates critical path delay. The problem is that we have no way of knowing actual wire delay until the very last (layout) stage of the VLSI CAD flow – only then can we identify slow connections in the design. We then optimize these slow connections through a variety of techniques, but unfortunately some of these techniques cause significant changes to the design and are hence very time consuming. Adding pipeline registers is a particularly effective technique; however, it may necessitate a major change to the design if its functionality was sensitive to the latency of each connection. The next time the designer compiles his/her design, there is no guarantee of timing closure either, creating a manual cycle of iterative improvement of design performance. This tedious time-consuming design → compile → repipeline cycle lowers productivity and is in many cases suboptimal.

Latency Insensitive Methodology

Carloni and Sangiovanni-Vincentelli proposed automating the insertion of pipeline registers to avoid the timing closure problem altogether [35, 36]. Their work essentially proposed independently designing the computation of a digital circuit and its communication. This separation of computation and communication allowed the independent optimization of either the compute modules or their communication architecture without changing the other. This enabled avenues for the automatic generation and optimization of the communication circuitry in a design.

Carloni and Sangiovanni-Vincentelli also proposed a concrete design flow to both enable and leverage the separation of computation and communication [35]. Figure 2.3 shows the main steps in the latency-insensitive design flow. The design is entered as a collection of IP modules connected together in a communication graph. This module-based design style was already the norm in VLSI systems since it promotes IP reuse and simplifies the design of large and complex systems; the only difference is that latency-insensitive design requires inter-module connections to be explicitly specified to differentiate

them from intra-module connections. The next step encapsulates each design module with a "shell" to make it tolerant to latency [35]. The shell has two functions: first, it adds stalling capability to the module to be able to stop its operation if all of its inputs are not ready, and second, it buffers the received inputs until all inputs are ready. The shell basically makes the module "patient" or insensitive to the latency of connections feeding it – it does not matter how many clock cycles it takes each connection to transfer its data to the module inputs, because the module will only accept the inputs when they are all ready. The final step in the latency-insensitive design occurs after placement, routing and layout of the design. Any slow inter-module connections are pipelined using *relay stations* to improve their delay. The relay stations are pipeline registers with the ability to stall if their downstream component is not ready to accept data [35].

Separate Computation and Communication

Latency-insensitive design succeeded in separating computation and communication, and allowed for the automatic improvement of communication delay with minimal changes to traditional VLSI CAD flows. Much research has since looked into the performance analysis and optimization of latency-insensitive systems [36, 102, 103]. Other work has added support for "soft connections" in `bluespec` (a hardware description language (HDL)) to allow the specification of latency-insensitive connections in digital circuits [121]. In the context of FPGAs, Murray and Betz refined the components of latency-insensitive design for FPGAs and quantified their overall cost [115]. Fleming et al. use latency-insensitive first-in first-out memorys (FIFOs) to abstract communication over multiple FPGAs [62]. In later work, Fleming et al. combine their FIFO communication abstraction with on-chip scratchpad memory and processor management to create a programming environment, called LEAP, for FPGAs [61].

A very good reference for further reading on latency insensitive design is in Carloni's retrospective paper "From Latency-Insensitive Design to Communication-Based System-level Design" [34]. In this paper, Carloni cites the principle of separate computation and communication as one of the important foundations upon which NoCs were created [34]. Indeed, much work has investigated the intersection of NoCs and latency-insensitive design. For example, Hemani et al. assert that the *"...overhead of [NoC] switching [should be] comparable to that of latency-insensitive design"* to be viable [72].

The communication in VLSI systems is now thought of as a separate entity that is designed and optimized independently from the computation. We believe that a NoC with a latency-insensitive communication protocol could potentially ease the design of FPGA systems and make it more efficient. We discuss the birth of NoCs in the following subsection, and survey relevant work in that field.

2.1.4 Networks-on-Chip

NoCs [29, 53, 88] have been proposed to overcome the complexity challenges faced by nano-scale electronic chips. An NoC is a dedicated communication substrate designed to transport data between IP modules using a latency-insensitive protocol[1]. NoCs typically structure the wiring of a circuit into a regular topology making it much easier to control its timing behaviour – this may eliminate timing closure iterations that were caused by the ad-hoc global connections between modules in a circuit [53]. Additionally, the regularity of NoCs facilitates modular design and promotes the use of standard interfaces which are both means to better design more complex systems [53].

[1]Some NoCs, such as circuit-switched variants, may have a fixed latency, but typical NoCs have a variable latency for transporting data.

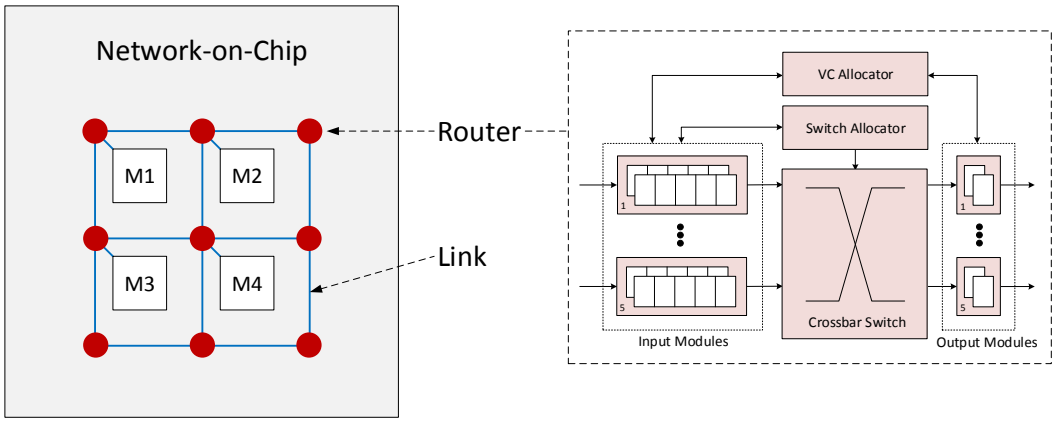


Figure 2.4: NoCs consist of routers and links. Links transport data and routers switch data between links.

NoCs draw their inspiration from macro-scale communication networks – both consist of routers and links as shown in Figure 2.4. However, NoCs exhibit different design tradeoffs because, unlike macro-networks, they do not have a stringent limitation on the number of router I/O pins but buffering is more costly. NoCs are intended for both (heterogeneous) multiprocessor-based architectures [53] and custom application-specific implementations of systems on a chip [29]. In the past two decades, NoCs have been both researched and commercially implemented [141]. In this subsection we summarize some of these findings divided into four main points of interest.

Router Microarchitecture

The architecture of NoC routers is heavily studied [52]. In the past decade, much work focused on improving the performance of NoCs, and reducing its area and power – we give three examples. Mullins et al. focus on reducing latency by optimizing the router control logic [112]. Kim uses crossbar partitioning [86] to reduce the switch area overhead. Additionally, clock gating was used in [111], and was found to considerably reduce dynamic power consumption.

An open-source state-of-the-art router implementation [54] incorporates many of the component variations of NoC routers. For instance, both separable and wavefront allocators [27] are implemented thus allowing the comparison of different microarchitectural choices. We use this router in our experiments and implementations in this thesis and we explain its detailed microacrchitecture in Chapter 3.

Modeling and Analysis

Through quick access to NoC efficiency/performance metrics, designers can better optimize NoCs in shorter design cycles; furthermore, these models are necessary for the automatic synthesis or optimization of NoCs by CAD tools. Most previous work constructed mathematical models for area and power based on the subcomponents of NoCs [25, 82, 144, 151]. In addition, C-based NoC simulators have been developed to quickly characterize NoC performance [80].

Some papers discuss the power breakdown of NoCs by router components and links, and investigate how power varies with different data injection rates in an NoC [70, 111, 144]. Such power-breakdown analysis guides the design and optimization of NoC components. Other work focuses on complete

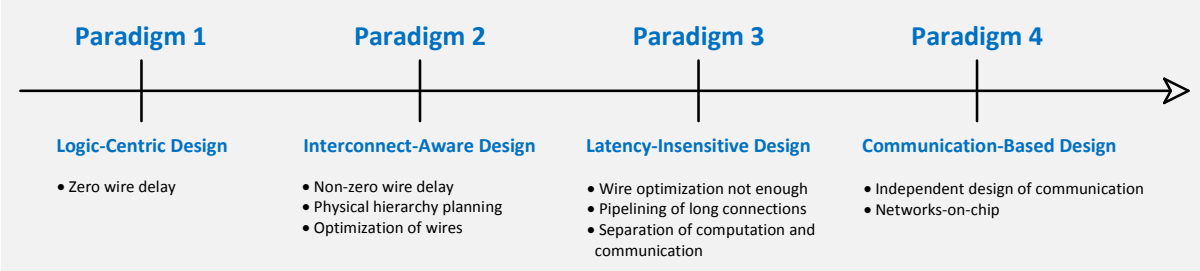


Figure 2.5: Four paradigms of interconnect design to handle the increasing complexity of VLSI circuits, identified in Section 2.1.

systems and reports the power budgeted for communication using an NoC [90, 135]. From a system design viewpoint, these measurements are crucial as power dissipation is becoming more and more of a limiting factor in dense submicron chips.

CAD Tools

There are many degrees of freedom in NoC design such as the topology, buffer size and number of virtual channels (VCs) to name a few. This has caused manually-designed NoCs to be overprovisioned and therefore unnecessarily large or power hungry; furthermore, the process of manually designing an NoC is time-consuming. This has motivated research into the automatic synthesis of NoCs [28, 67, 76, 77, 113, 114, 116]. Of special interest is the problem of mapping applications to regular NoC topologies. While some prior work uses a branch and bound algorithm to map an application to an NoC [77], other work models the mapping as a quadratic assignment problem and solves it using tabu search [113], or uses simulated annealing [105]. Sahu and Chattopadhyay provide a good survey of the numerous different mapping algorithms for NoCs [127].

Comparison with Buses

Because NoCs are proposed as an alternative to multiplexed buses and point-to-point connections, much previous work has compared these three interconnect options [107, 130]. Comparing different types of interconnects is tricky because they exercise different tradeoffs; for example, a bus may have lower bandwidth and lower power compared to an NoC so comparing either bandwidth or power by themselves may give an incomplete picture. This is why some previous work has used architecture- and application-independent metrics, such as the energy-per-bit, to compare different kinds of interconnect [22, 23]. This captures both the power and the bandwidth of an interconnect in a representative efficiency metric for comparison.

Buses are the suitable interconnect for small systems and some researchers have investigated the question of when NoCs will become the preferred option [152]. By increasing the number of cores connected in a system, and measuring the frequency and latency of different traffic patterns we can quantify the point at which NoCs become more efficient, or better performing than buses [152]. Some work focused on exploring a single real application when using either an NoC, a bus or point-to-point links for interconnection [94], while other research focused on making synthetic traffic more realistic to more accurately compare the performance of NoCs and buses [142].

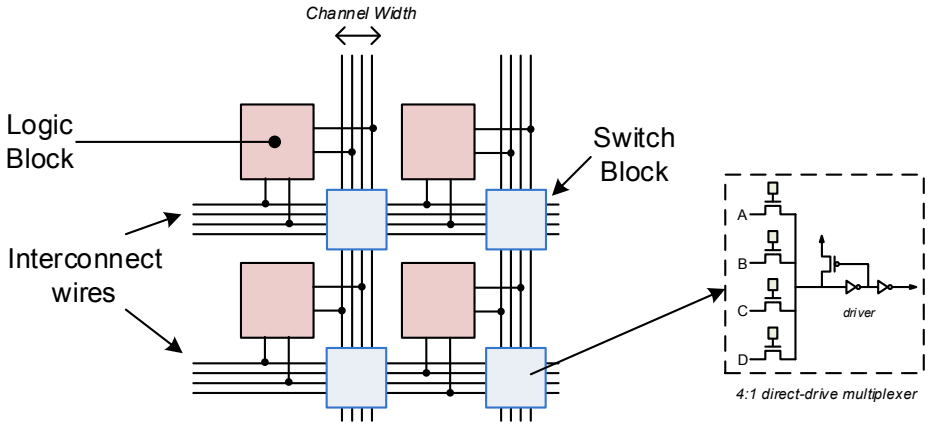


Figure 2.6: Island-style FPGA architecture logic and switch blocks. A sample pass-transistor-based implementation of a 4:1 routing multiplexer is shown.

2.2 FPGA Interconnection

In the past two decades, VLSI circuit design has exhibited a paradigm shift to handle its increasing complexity and capacity, from logic-centric design to communication-based system-level design [34]. In Section 2.1 we showed how design is passing through four stages in this shift towards communication-based design using NoCs (summarized in Figure 2.5). In this section we focus on FPGAs. We investigate how FPGA interconnect architecture changed in the past two decades to accommodate the higher wire delay and design complexity. We also motivate the addition of embedded NoCs by looking at current FPGA system-level interconnects and problems. We discuss how traditional soft buses make it challenging to connect to fast I/O interfaces, and are a barrier to much-needed modular design on FPGAs. Finally, we discuss how an embedded NoC can make design more modular on FPGAs, and we show that this is important and relevant for many FPGA design domains.

2.2.1 FPGA Logic and Interconnect Scaling

Figure 2.6 shows an FPGA's fabric at a high level. The logic blocks contain lookup tables (LUTs) and registers that can implement any digital logic function. The programmable interconnect consists of wires and switch blocks – these are used to connect logic blocks together to create useful functions. Switch blocks allow programmable interconnection of the routing wires, and are now most commonly implemented with "direct-drive" multiplexers, which use pass-transistors followed by a buffer and driver as shown in Figure 2.6 [97].

FPGA Logic Scaling

We look at the past two decades of FPGA development to investigate the trends in device architecture. Figure 2.7 shows the ratio of soft logic, I/Os, memory and hard IP in Altera FPGAs and illustrates how FPGAs have continuously added hard blocks to augment the soft fabric. First, embedded BRAM were added to implement more efficient on-chip storage. Second, DSPs were added to implement arithmetic operations more efficiently. Next, FPGA vendors started to embed I/O interfaces such as memory controllers and PCIe interfaces due to their widespread use and efficiency when implemented hard.

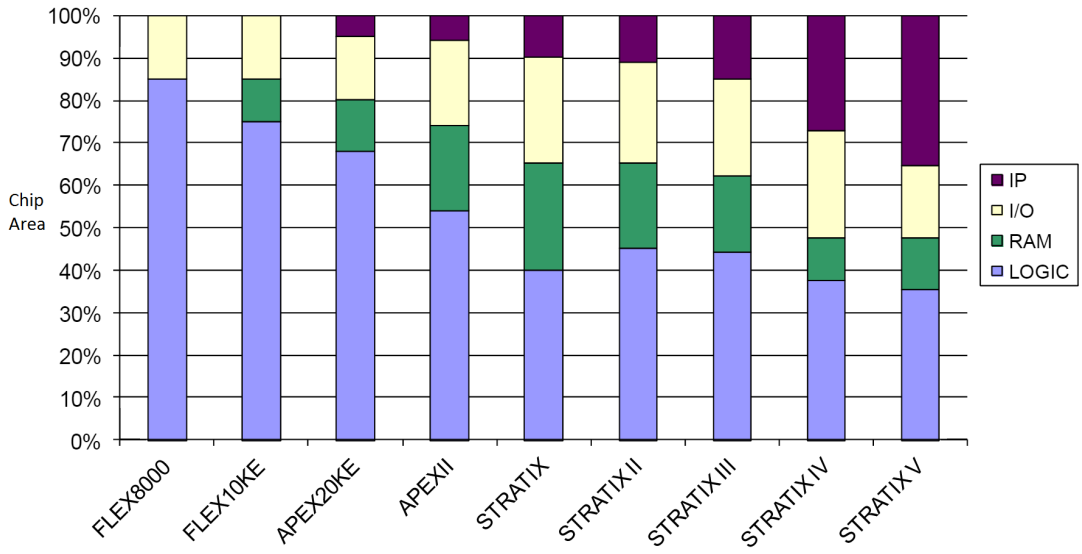


Figure 2.7: Trend in Altera FPGAs to augment fine-grained fabric with hard blocks (from [33]).

Recent devices also contain multi-core embedded processors and the DSPs have been upgraded to support single-precision floating-point arithmetic operations [91]. Even though the fraction of the die devoted to soft logic decreased over the past 20 years, the absolute number of logic blocks increased exponentially. Stratix I [97] was released in 2003 and Stratix 10 [99] in 2016, with Stratix II–V in between [95, 96, 98]. Between Stratix I and Stratix 10, the number of logic *elements*[2] grew from ~45,000 to ~3,000,000 – a ~65× increase in soft logic capacity.

FPGA Interconnect Scaling

Between Altera's Stratix I and Stratix V there were no fundamental changes to the FPGA's interconnect. One only needs to look at the publications about the Stratix-series FPGAs to confirm this. In the Stratix II paper, the authors say that *"Stratix II uses a routing architecture that is similar to Stratix."* [95]. The Stratix III-IV paper focuses on power management techniques and the embedded memory blocks, but does not mention the interconnect architecture at all [96]. In the Stratix V paper, the authors *"...explore minor variations [in the interconnect architecture] that could keep pace with the increase in routing demand as well as obtain performance improvement."* [98]. The "minor variations" that the FPGA's interconnect underwent comprise of changes in channel width, wire length and buffering to work best with each device architecture. Xilinx FPGAs have not undergone any major changes to their interconnect either. In their latest paper, Xilinx' FPGA architects identify the problem of wire scaling, and their current solution is to *"...aid placement and routing algorithms to achieve critical paths with fewer interconnect resources"* [40].

Murray and Betz show the FPGA's interconnect frequency has improved modestly for local communication covering the same logical distance, but deteriorated for global communication that spans a constant physical distance [115]. Indeed, interconnect consumes a large portion of the critical path delay on FPGAs because each connection consists of both wires and pass-transistor multiplexers [44]. However, designs are typically heavily pipelined on FPGAs due to the abundance of on-chip registers – as

[2]FPGA vendors measure logic capacity in logic elements which roughly refer to a 4-input LUT and a bypassable register.

long as the register-to-register delay was still acceptable, FPGA frequency could scale slightly from one generation to the next. This is reminiscent of the interconnect-aware design paradigm (see Figure 2.5) where the optimization of wires was enough to maintain scaling.

Only in Stratix 10 was wire optimization no longer sufficient for interconnect scaling. Stratix 10 FPGAs include pipeline registers within its interconnect fabric (originally proposed two decades prior by Ebling et al. [56], and Singh and Brown [136]) to enable finer-grained and lower-cost pipelining of connections [99]. Furthermore, the architects of Stratix 10 acknowledge that the design style can no longer remain logic-centric, and that users will *"...perform some work to make [the design] amenable to pipelining."* [99]. By adding registers to the interconnect fabric, pipelining becomes easier and more effective. This combination of a pipeline-amenable FPGA architecture and the change in design style indicate that FPGAs are moving to the third design paradigm of latency-insensitive design (see Figure 2.5). However, communication and computation are still coupled in the FPGA fabric since they are both synthesized, placed and routed in the same monolithic CAD flow. This coupling hampers modularity and the potential for independent design, optimization and implementation of IP blocks and the interconnect.

Ye and Rose pointed out that the fine-grained FPGA interconnect may not be suitable for wide datapath circuits [150]. In their 2006 paper, they modified the FPGA architecture to partially use multibit routing so that multiple wires are controlled through a single configuration random access memory (RAM) cell [150]. They show that they can improve FPGA area by ~10%, only for datapath-heavy circuits [150]. Their results emphasize the need for a coarser-grained interconnect to better-suit the ever-more prevalent wide-datapath circuits that are implemented on FPGAs.

Our main goal in this thesis is to augment modern FPGAs with an NoC to decouple communication and computation altogether, and move towards the paradigm of communication-based design. Because FPGAs are smaller than ASIC VLSI circuits by ~35× , they lag 4 or 5 generations behind in capacity [89]. This is why we believe that even if it is not required now, an advanced communication-based FPGA architecture will become urgent in the future to (at least) handle the increasing design complexity. Additionally, a communication-based architecture – such as one based on a coarse-grained NoC – can be designed so as to better map wide-datapath applications onto FPGAs.

2.2.2 FPGA System-Level Interconnect

In this section we present buses: the de facto standard of connecting systems on FPGAs. We focus on the connection to fast I/O interfaces, a common and very important sub-problem in FPGA application design, which is increasingly difficult to do with soft buses. Additionally, we explore the need for modular design on FPGAs, motivated by FPGA compute acceleration, high-level synthesis (HLS) and partial reconfiguration. We believe that a dedicated system-level interconnect (such as an embedded NoC) will effectively decouple the design of each compute module in an application, making FPGA application development more modular and therefore easier, faster and more scalable.

Buses for System-level Interconnection

Figure 2.8 is a block diagram of a bus connecting multiple masters to multiple slaves. The bus consists of a large, centralized multiplexer controlled by an arbiter. Additionally, a bus will often contain pipeline registers at various locations to improve its speed [85]. If modules connecting to the same bus operate

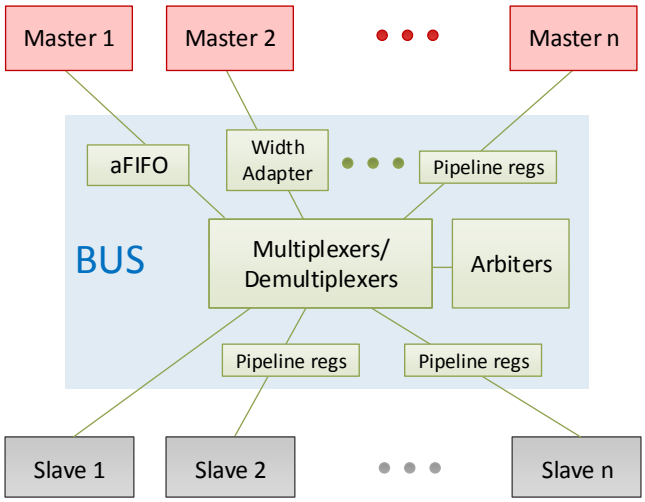


Figure 2.8: A block diagram of a typical bus connecting multiple masters to multiple slaves.

at different frequencies, the bus must also contain clock-crossing circuitry such as an asynchronous first-in first-out memory (aFIFO) to bridge clock domains. Similarly, if different-width modules connect through a single bus, width adapters are required to adapt the width accordingly. This style of bus-based interconnect is used by both Altera and Xilinx in their system-integration tools [49, 79].

System-level interconnection tools allow the designer to focus on designing the compute modules, and then the tool automatically connects the modules together to match a detailed designer specification. While this greatly helps the design of complex systems, the system integration tools do not solve the problems that are inherent to soft buses. In academia, Orthner implemented a tool that creates soft buses for system-level interconnection [85]. His work focused on automating the creation of multiplexer-based buses for implementing transaction communication. More recently, Rodionov et al. worked on GENIE – an academic interconnection tool that targets both fine-grained and coarse-grained connections [125, 126]. Rodionov et al. use a split/merge NoC since it is distributed and more scalable than centralized multiplexer-based interconnects. This trend of moving from centralized bus-based interconnects to distributed NoC-based ones is evident for both FPGAs and ASICs.

Soft bus frequency is unpredictable making timing closure difficult and often requiring manual designer intervention in designing the bus itself. Additionally, soft buses consume much FPGA area, power and compilation time since they use the FPGA's soft fabric. In their physical implementation, there is no clear boundary between a system's compute modules and the system-level interconnect bus, consequently, the soft bus creates an undesirable coupling between compute modules. For example, the physical size and placement of the bus will influence where compute modules are placed on the FPGA. Additionally, timing paths exist between modules through the bus, which might force all components of a system to be co-optimized and compiled together in a slow and monolithic CAD step, thus hampering efforts to individually compile modules in a system. A prefabricated embedded NoC, as presented in this thesis, can decouple modules more effectively and greatly enhance modular design and CAD.

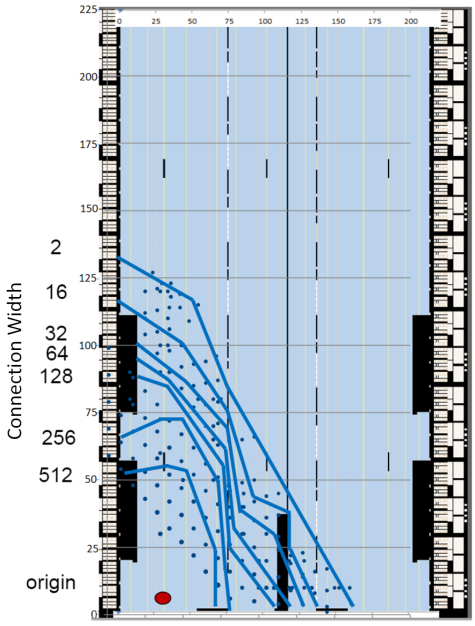


Figure 2.9: Physical reach of different size connections (2–512 bits) at 400 MHz (from [24]).

Connecting to I/O Interfaces

Connecting to I/O interfaces is among the most important functions of a system-level bus wherein it distributes I/O data to/from external memory and fast transceivers. A prime example of such a bus is formed between a double data rate version 3 (DDR3) memory controller and the multiple masters that access it. A typical DDR3 controller can run at 1067 MHz (and both clock edges) and 64 bits; this must be downconverted (at quarter-rate) to ~267 MHz to operate at the FPGA fabric speed, and hence requires a bus that is 512 bits wide. This bus often spans a large portion of the FPGA chip to connect the DDR3 controller and the multiple masters accessing its data. Its large size makes it both difficult to design and inefficient.

- **Difficult design**: This difficulty stems from the timing closure requirements. This bus must be designed so that its frequency is high enough to keep up with DDR3 bandwidth – 267 MHz in our example. Designing a bus that is both very wide and very fast is difficult on FPGAs and often requires the addition of multiple pipeline stages. This hurts productivity as the designer must compile the design (a multi-hour process for a large FPGA), check if the frequency target is met, and if not take corrective action like adding pipeline registers and recompiling again (timing closure problem). In his 2016 talk [24], Greg Baeckler highlights this problem by showing the reach of different-size connections on a modern FPGA. As Figure 2.9 shows, wide 512-bit buses can only reach about one fifth of the FPGA when running at 400 MHz. This is due both to the accumulated delay of successive short wires stitched together in the FPGA's interconnect and to routing congestion.
- **Hardware and CAD Inefficiency**: These wide/fast buses use much of the FPGA's soft logic and interconnect because of their large width. They also lead to a large Computer-Aided Design (CAD) problem, as the CAD system must synthesize, place, and route each LUT and register that together compose the bus; this exacerbates the long compile time of FPGA tools.

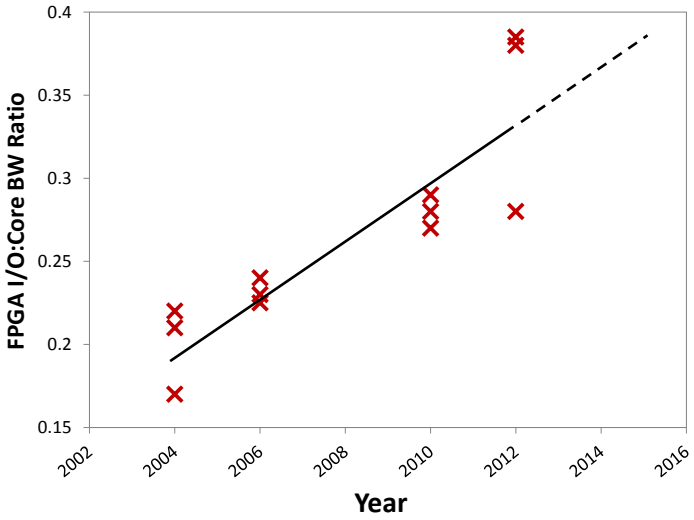


Figure 2.10: The ratio of I/O transceiver bandwidth and core bisection bandwidth across FPGA generations. Stratix I-V data for the 3 highest bandwidth devices in each generation.

Hence, a major part of the challenge of system-level interconnect design is due to the construction of a coarse-grained wide connection using very fine-grained soft logic, begging the question of whether it can be done in a better way. The FPGA's soft logic and interconnect is very flexible and capable of creating highly customized circuits. However, FPGA architects have already realized the need for more efficient hard logic (multipliers, adders and processors), hard memory (BRAM) and hard I/O (embedded I/O controllers) – these hard blocks process wide data much more efficiently than the soft fabric. Similarly, this thesis proposes an embedded interconnect to augment the FPGA's fine-grained fabric, and to implement wide connections – that do not require fine-grained configuration – more efficiently.

Figure 2.10 plots the ratio of FPGA I/O transceiver bandwidth to the bisection bandwidth of the FPGA on-chip interconnect; the I/O transceiver bandwidth has been growing at a faster rate than the bandwidth of the FPGA's programmable interconnect. This highlights that moving all the I/O data across an FPGA with the traditional interconnect is becoming ever more challenging and further motivates the addition of a high-bandwidth on-chip interconnect to augment the existing fabric and help keep up with the increasing I/O bandwidth demands.

The Need for Modular Design on FPGAs

An FPGA application consists of compute modules and a system-level interconnect for communication between the modules. Much work has been done to more easily create the compute modules on FPGAs; however, system-level interconnection was often a secondary issue. This is similar to how the FPGA architecture itself has increasingly included hard blocks (such as processors, DSPs and BRAM) to enhance computation, whereas the interconnect has seen less radical change. Improving computation is undoubtedly important for FPGA applications; however, we argue that the increase in design complexity necessitates a focus on on-chip communication as well. This section enumerates different ways in which the design of compute modules was made easier and more modular on FPGAs, and draws parallels for system-level interconnects. Other than easing timing closure, we show how an embedded NoC

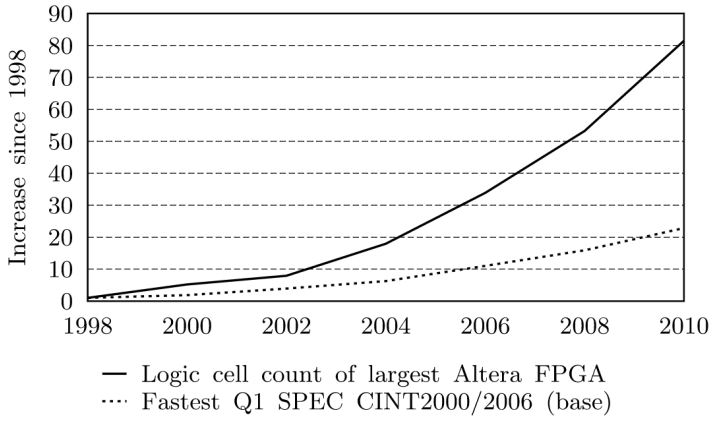


Figure 2.11: FPGA logic capacity and processor speed over time. FPGA size is increasing at a much faster rate making CAD compilation time a challenge (from [106]).

can complement the advances in FPGA systems towards more modular communication-based FPGA systems.

Compile time: Long compilation times are a huge problem in FPGA CAD. Ludwin and Betz highlight this problem in Figure 2.11 by showing that FPGA size is increasing at a faster rate than processor speed thus requiring big CAD flow innovations to maintain the same runtime [106] – particular emphasis was placed on making CAD steps more parallel [20]. From a design perspective, Lavin et al. proposed pre-compiling "hard macros" to speed up compilation of compute modules [92], but no work tackled the compilation of the system-level interconnect. A hard NoC can eliminate the compile time of the system-level interconnect since it is prefabricated in hard logic. Furthermore, modules communicating through the NoC will be disjoint and timing-decoupled enabling their parallel and independent compilation which may considerably improve CAD runtime and design modularity.

High-level Synthesis: Hardware designs are time-consuming and difficult to specify partly because they are written in low-level HDLs. HLS tackles this problem by compiling easy-to-use programming languages (such as C/C++) to hardware [63] – this makes the design of compute modules easier but does not tackle system-level interconnection. The existing system interconnection tools are then used to creates buses to connect the HLS compute modules in a system – we propose replacing the existing tools with ones that leverage embedded NoCs instead, thus alleviating the limitations of soft buses.

Data Center Computing: An important application of FPGA computing is the acceleration of data center applications such as Microsoft's acceleration of their search engine [122]. Additionally, other applications have been accelerated using FPGAs related to geoscience [101], financial analysis [45], video compression [42] and information filtering [41] while demonstrating order-of-magnitude gains in performance and power efficiency compared to multicore processors and graphics processing units (GPUs). In such systems, an important part of the FPGA system is the "communication infrastructure" (or "shell"), which connects the application compute kernels to each other and I/O interfaces – Figure 2.12 demonstrates the anatomy of a typical FPGA compute accelerator. Currently, the design of the shell using a custom soft bus is a painstaking exercise riddled with timing closure iterations because it typically connects to many fast I/O interfaces. Additionally, shells are not portable so they need to be redesigned for each FPGA device. Instead, an embedded NoC can abstract communication to I/Os and make shells easier to design, more portable and more efficient.

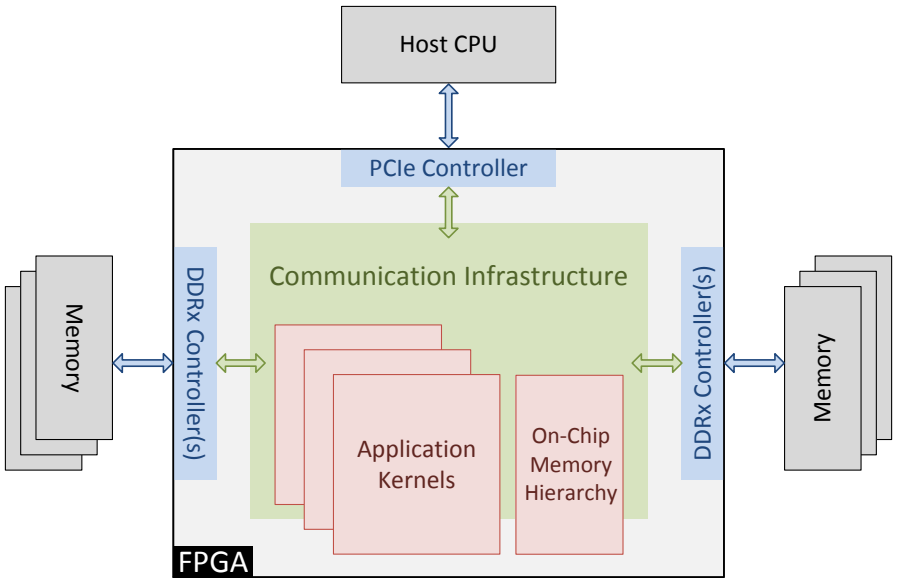


Figure 2.12: Architecture of a typical reconfigurable computing system. An FPGA is connected through PCIe to a host processor and through DDRx controllers to external memory. Communication Infrastructure refers to the interconnect that is designed to connect the application kernels to external devices as well as on-chip memory.

Partial Reconfiguration: One of the main barriers to partial reconfiguration on FPGAs is the complexity in creating the interface to a newly-configured module, and connecting the new module to the rest of the (running) FPGA application. Both Xilinx's and Altera's partial reconfiguration flows lock down all interfaces to partially reconfigured modules [19, 147], and these interfaces are of fixed size and protocol. It is often the case that the system-level interconnect (connecting partially-reconfigurable modules together) is also locked down [19, 147]. An embedded NoC can become a prefabricated interconnect and interface to natively support partial reconfiguration in FPGAs. A newly-configured module can connect to an NoC port and immediately have access to communicate with the rest of the design. By decoupling the communication between modules in a design, an embedded NoC can more easily handle the swapping of modules in a partial reconfiguration system.

2.3 FPGA-Based NoCs

We motivated the need for embedded NoCs to handle the ever-increasing design size and complexity of FPGA applications. To put our embedded NoC proposition in context, this section summarizes important existing work on hard and soft FPGA-based NoCs.

2.3.1 Soft NoCs

Soft (overlay) NoCs have been extensively studied in the context of FPGAs, motivated by improved design modularity, better scalability and the use of standard interfaces. Some work used NoCs to implement multiprocessor systems on FPGAs [55, 128, 129]. However, most other work sought to create a general-purpose NoC for use as a system-level interconnect in FPGA applications – this work focused

mainly on creating an efficient NoC implementation that is suitable for the FPGA fabric. In this section, we comment on some of these soft NoC implementations and compare them against each other.

Switching Methods

Motivated by area efficiency and design simplicity, different switching methods are implemented on FPGA NoCs. PNoC uses circuit switching because that minimizes hardware overhead due to the absence of buffers and the simple control logic [74]. Store-and-forward switching buffers a whole packet at each router before proceeding to the next hop. This is used in LiPaR because it greatly simplifies the arbitration logic saving on the arbitration delay, area and power compared to wormhole switching [133]. Virtual cut-through switching also requires support of buffering a full packet but, unlike store-and-forward, packets may progress to the next hop even if the entire packet is not yet buffered at the current router. TopAd uses virtual cut-through switching motivated by long packets which are unsuitable for plain wormhole switching [26]. Wormhole switching allows flits to proceed to the next hop even if there is not enough space to buffer the entire packet. This reduces latency further but may lead to poor link utilization for long packets distributed across multiple routers due to head of line (HOL) blocking. CONNECT [117], OpSrc [57], RASoC [153] and NoCem [131] are all wormhole NoCs that implement switching decisions based on a single flit rather than a whole packet, thus reducing buffering and improving throughput at the cost of more complex arbitration logic in each router.

Virtual Channels

A VC is a separate buffer queue at each router input port [120]. This can be used together with any switching method to increase throughput by alleviating the HOL blocking problem. If a packet output is blocked, another packet can proceed through the same physical link on a separate VC. Schelle and Grunwald compared an instance of NoCem with no VCs to another with two VCs and reported more than twice the performance for a high-throughput communications application [131]. TopAd does not use VCs but its authors plan to implement a wormhole-switched network with VCs in their next iteration to maximize throughput [26]. CONNECT is a VC router than can be parameterized to include any number of VCs; it boasts comparable throughput to a state-of-the-art ASIC-intended router implemented on an FPGA [117]. Huan and DeHon argue that a radically different network architecture that uses no VCs better suits FPGAs [78, 84]. They replace routers with split and merge primitives and heavily pipeline their NoC, resulting in a 2–3× performance improvement over CONNECT [78]. Recently, split and merge blocks were used in an automated CAD tool to generate interconnect for FPGAs [125, 126].

Routing

Routing defines the path taken through the network. Adaptive routing takes network traffic into account by favoring low-utilization links while deterministic routing will always send a packet on a predetermined path regardless of congestion. Most FPGA NoCs use deterministic routing because of its low complexity and hardware overhead: a simple implementation uses a routing table in each node that tells a flit where to go next. This is used in CONNECT [117], OpSrc [57] and PNoC [74]. Routing tables grow quadratically with the number of network nodes so they do not scale for large network sizes and can be replaced with simple routing logic as implemented in LiPar [133], NoCem [131] and RASoC [153]. TopAd implements operating system (OS) controlled adaptive routing combined with routing tables [26]. The

OS is continuously gathering traffic statistics on a dedicated low-cost NoC, then updates the routing tables in the data NoC routers such that traffic is evenly distributed [26]. This kind of adaptivity requires an OS and is less responsive to traffic variations compared to hardware-controlled adaptive routing which was very scarce in the FPGA literature. Hoplite uses adaptive deflection routing in a very light-weight NoC architecture [83]. By removing all buffering and optimizing the crossbar, Hoplite is *tiny* compared to other NoCs, however, it is restricted to single-flit packets and suffers from a very high data transmission latency in adversarial traffic patterns as a result of deflection routing [83].

Flow Control

Flow control specifies when a packet/flit can proceed to the next hop based on buffer space availability [120]. LiPar [133] and RASoc [153] use request/acknowledge handshake signals. A common method that is used in OpSrc is on/off flow control which stops packet transmission when a buffer is almost full [57, 84]. CONNECT uses credit-based or peek flow control, both keep track of buffer space availability in downstream ports [117]. Peek flow control has dedicated wires to relay buffer space availability, making use of the ample FPGA routing resources [117].

2.3.2 Hard NoCs

One of the main barriers to the adoption of soft NoCs is their large area overhead and poor operating frequency. A 16-node, 128-bit CONNECT NoC, for instance, uses 36% of a Xilinx LX240T FPGA LUTs and can run at ~113 MHz – a 1.5× improvement compared to an NoC that is not tailored to FPGAs [117]. However, if we want to use this soft NoC to distribute data from fast external memory (for example, a 64-bit, 800-MHz DDR3 memory controller), it would need to be 4× wider and 2× faster to transfer DDR3 bandwidth across its links at quarter rate, which would almost consume the entire area of the LX240T FPGA. We argue that realistic high-bandwidth FPGA applications will not benefit from a soft NoC. These very applications are the ones that are difficult to design because of their high complexity, connections to fast I/O interfaces and tough timing closure. This has led research into hard NoCs where an NoC can ease the design of a complex FPGA application with much lower overhead and higher performance. However, work on hard NoCs has been somewhat scarce (which partly motivated the work in this thesis). In this section, we outline previous work that proposes the addition of a hard NoC to FPGAs.

Early Work

In 2002, Marsecaux et al. were the first, to the best of our knowledge, to propose adding a hard NoC to FPGAs [109]. Marsecaux et al. leveraged FPGAs to implement an extensible processor that can dynamically configure and use custom hardware accelerators (for tasks such as encryption or image compression) [109]. They used a hard NoC, to manage communication between the processor and the dynamically reconfigurable hardware accelerators. The hard NoC uses wormhole packet switching to reduce the amount of required buffering, two VCs to avoid deadlock, and standard interfaces to the accelerators called "net cells". For their prototype, a 16-bit, 4-node hard NoC occupied 34.8% of their FPGA. In 2003, Marsecaux et al. refined their system, and proposed including three hard NoCs (for configuration, data and control) to properly synchronize messages between the operating system and accelerators [108].

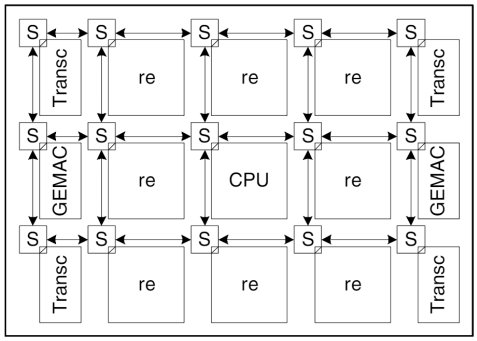


Figure 2.13: Early work proposed a hard NoC that connects both the core and I/O of FPGAs (from [71]).

Marsecaux's early work attempted to create a very specific hard NoC that only works for their reconfigurable processor. Because of FPGA size at the time, the overhead for a small NoC that only supports dynamically-reconfigurable systems – a niche application of FPGAs – was prohibitively expensive. Hecht et al. addressed these shortcoming by extending the NoC to work both for dynamic reconfiguration and as a general system-level interconnect [71]. They assert that *"...a packet or message based inter-module communication with common interfaces improves design reuse and facilitates a building-block-based design especially in the case of network processing."* [71]. Figure 2.13 shows their proposed NoC that connects both the FPGA's core and I/Os. After presenting the *idea* of a hard NoC, Hecht et al. presented a SystemC simulation model of a hard NoC to enable future research on the topic [71].

In their 2007 paper, Gindin et al. propose a semi-hard NoC with only part of the router implemented in hard logic, but the network interface in charge of routing was left soft to maximize reconfigurability [66]. Additionally, they propose dividing the FPGA into physical tiles, which either consist of "configurable" LUTs or "fixed" hard blocks, but not both together – this was mainly done to break-down the compilation of a large design into smaller manageable pieces. Gindin et al. only discussed the vision of their hard NoC without any details of the architecture, then focused their paper on implementing a routing algorithm that is suited to an FPGA NoC [66].

NoC to Unify Configuration and Data

In 2008, Goossens et al. proposed to radically change the FPGA architecture by adding a hard NoC that is both for configuration and data transport [69]. They aim to improve both global communication and dynamic partial reconfiguration by using the same NoC. To quantify the overall overhead of this NoC, the authors assume each 1000 LUTs – a reasonable IP module size – were connected to a NoC router, in which case a hard NoC would occupy 14% of an FPGA's area [69]. Additionally, a 32-bit hard NoC router provided 149× better performance per area compared to a soft implementation thus motivating the need for a hard implementation [69].

In their paper, Goossens et al. detail how their NoC configures the FPGA and handles data movement; for example, they implement latency guarantees by using *"...a virtual-circuit TDMA scheme"* [69]. However, the impact of the proposed radical change to the FPGA fabric is unclear from their analysis. How does the new FPGA holistically compare to a conventional FPGA? Additionally, the authors did not discuss the CAD system for their proposed FPGA. The complicated coupling of configuration and

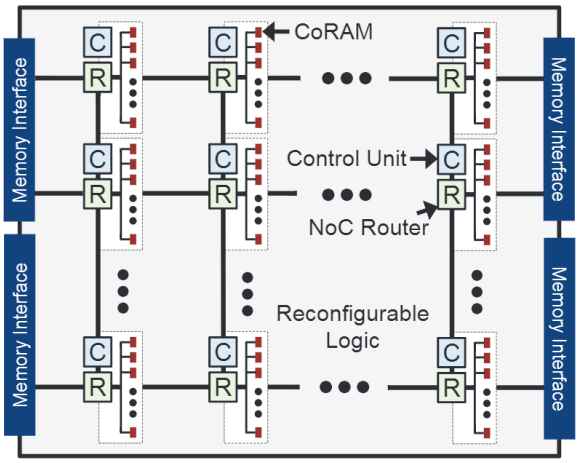


Figure 2.14: CoRAM architecture uses an NoC to abstract communication to on-chip and off-chip memory (from [46]).

data movement is likely to make both the creation and use of a CAD system very difficult. However, one of the propositions set forth in Goossen's paper was adopted by FPGA vendors eight years later – configuration in Altera's Stratix 10 FPGAs is now done using a *dedicated* NoC [104].

Time-Multiplexed FPGA Wiring for NoCs

Francis and Moore propose changing the traditional FPGA interconnect architecture to better-suit the implementation of hard NoCs. Their key observation is that a hard NoC will run faster than the FPGA's soft logic, consequently, they propose time-multiplexing the FPGA's interconnect to run as fast as hard blocks [64, 65]. In their first paper, Francis and Moore focus on changing the traditional FPGA interconnect to include time-multiplexed wiring, and they develop a scheduling algorithm to statically schedule a design's wires in time slots on a physical time-multiplexed wire [65]. By doing so, they were able to reduce the demand for FPGA interconnect wiring by ~70-80%. However, their results are likely optimistic due to unrealistic circuit assumptions; for example, they assumed that the time-multiplexed FPGA interconnect (at 90 nm) is bidirectional and can run at 2 GHz [65].

In their second paper, Francis and Moore propose a hard NoC in which the router connects through time-multiplexed wiring [64]. They compare the area of hard and soft NoCs implemented on both conventional and time-multiplexed FPGAs and find that 32-bit NoCs occupy less than 8% area of an FPGA, with hard NoCs ~4× smaller than soft [64]. While the results seem promising, their limited scope (only 32-bits and 8-bits were studied) and unclear methodology (the paper did not state how the area numbers were computed) make it difficult to draw detailed conclusions. However, Francis and Moore's work was the first to use time-multiplexing to leverage the speed gap between a hard NoC and the slower FPGA fabric.

CoRAM

In their 2011 paper, Chung et al. used NoCs to abstract memory access on FPGAs [46]. Their goal was to create a memory architecture (called CoRAM) such that FPGA applications can easily access memory words through standard "load" and "store" commands without knowledge of the underlying memory

organization and hierarchy. This would greatly ease the design of FPGA computing applications since it automatically creates the memory hierarchy for an application and presents a simple interface, thus relieving the burden of memory management from the FPGA application developer.

Figure 2.14 shows the envisioned architecture of CoRAM [46]. An NoC connects both external and on-chip memory resources, wherein a control unit at each router executes memory operations. The NoC transfers memory requests from the control units to the memory resources, and sends back the correct data response [46]. Chung et al. advocate the use of a hard NoC for its efficiency and performance advantages over soft NoCs [46]. In later work, Chung et al. prototype the CoRAM memory architecture using hard NoCs and they quantify its area overhead to 1.7% of a modern FPGA [47].

2.4 Summary

This chapter summarized some of the prior work relating to interconnect research in VLSI design. We started by investigating how VLSI circuit design progressed from a logic-centric methodology, to a communication-centric one over the past few decades. The increase in complexity and size of VLSI systems spelled the end of the scaling of long chip-wide wires. This has ultimately led to the separate design of circuit behavior and communication, where the latter was often implemented as an NoC.

Our investigation of NoCs informed the NoC architectural choices and evaluation methodologies that are used later in this thesis. We found that VC packet-switched routers are most high-performance and versatile in dealing with different applications. We also outlined existing NoC modeling and CAD tools; one of which (Booksim) we used in simulating our NoC. A survey of NoC versus bus comparisons informed the methodology and metrics used when comparing an embedded NoC to soft buses.

The second part of this chapter focused on FPGA interconnection. We emphasized that the FPGA interconnect has not changed significantly, even though the FPGA's logic has been constantly upgraded over the past two decades. We then explained the current system-interconnection challenges of FPGAs and motivated the need for more modular system-level design. Finally, we focused on FPGA-based NoCs, where we provided a summary of soft NoCs, and then delved into details of hard NoC proposals on FPGAs. Prior work on hard NoCs on FPGAs was scarce and often lacked sufficient architectural and evaluation details – this is part of the motivation for work in this thesis. We found that, like ASICs, system-level interconnection on FPGAs could benefit from an NoC. In the remainder of this thesis, we propose embedding NoCs on FPGAs, and we investigate the architecture, applications and CAD of such an embedded NoC.

Part I

Architecture

Table of Contents

Chapter 3

Router Microarchitecture

Contents

Our first step in evaluating NoC use on FPGAs is to determine what parts of the NoC should be hard (implemented in dedicated silicon) versus soft (implemented from the existing FPGA fabric). To that end, we study hard and soft NoC implementations on FPGAs to understand and quantify their efficiency versus flexibility trade-off. In doing so, we need to evaluate different NoC architectural options to find a suitable implementation for FPGAs – this requires hardware design of the NoC routers. Instead of designing the hardware ourselves (a research project in itself), we use an open-source state-of-the-art implementation of NoC routers developed at Stanford University in the Computer Architecture Group [54].

The Stanford router is full-featured and parameterizable, and supports different implementation options for its components. We can vary the NoC link width, number of VCs, buffer depth per VC and number of ports among other important router parameters. Additionally, the router includes multiple implementation variations for its subcomponents, such as different types of allocators, different buffer organization schemes and different crossbar implementation options. We use this chapter to present and justify the implementation options and parameters that we chose. Additionally, we explain the microarchitecture of each router subcomponent as these details are important for understanding the router's operation, and the design tradeoffs between their hard and soft implementations.

3.1 Routers

We use a parametrized open-source state-of-the-art VC router [54]. We focus on this full-featured router for two reasons. First, as FPGAs increase in capacity and hence contain larger applications, it will often

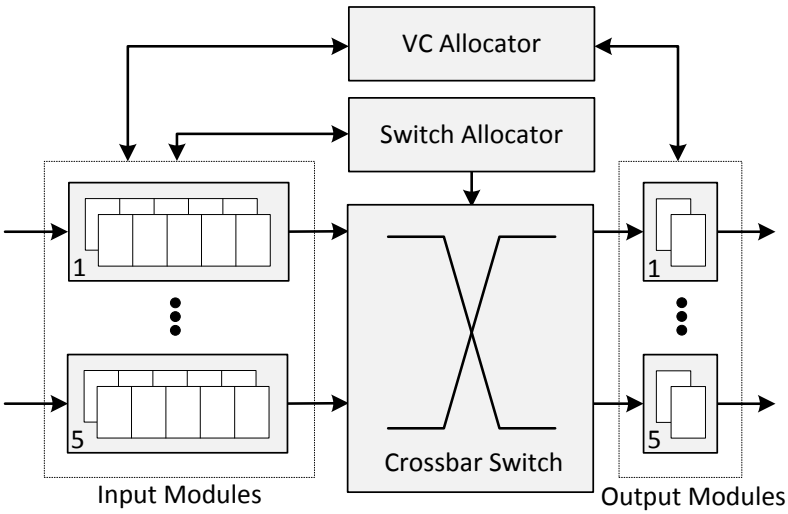


Figure 3.1: A VC router with 5 ports, 2 VCs and a 5-flit input buffer per VC.

be necessary to traverse many network nodes to communicate across the chip, and consequently we expect routers with low latency, such as the chosen router, to be important. As we show in Chapter 5, this router can also achieve high clock frequencies, helping it keep up with the throughput demands of the high bandwidth I/O interfaces on modern FPGAs. Second, since our focus is quantitatively examining the speed and area of hard and soft implementations of a wide variety of NoC components, use of a more full-featured router with more components yields a more thorough study; simpler routers would use a subset of the components we study. For example, a router that does not support VCs will not contain a VC allocator, but it is included in our study for completeness.

Instead of sacrificing NoC performance to adapt routers to FPGAs, we focus on using a complex but feature-rich router because of the following performance and flexibility advantages:

1. VCs: Other than their performance benefit, we believe that the flexibility of VCs will be beneficial for a reconfigurable platform such as FPGAs. We leverage VCs to adapt our NoC to FPGA design styles in Chapter 8.

2. Low latency: Much of the optimizations in this router ensure that it has a very low latency which is desirable on FPGAs where the designer typically has fine-grained control over every cycle of latency in their design.

3. High frequency: Other optimizations in this router include carefully-written hardware code that allow this router to operate at a high frequency, improving both bandwidth and overall latency, as this high frequency is obtained with a fairly short routing pipeline.

The router operates in a 3 stage pipeline that can be reduced to 2 stages if speculative VC allocation succeeds [54]. Ingress flits are stored in the input buffers and immediately bid for VC allocation; this is followed by switch allocation and switch traversal. When speculation is successful, both VC and switch allocation occur in parallel and thus saves one clock cycle of latency. Lookahead routing is done in parallel to switch allocation and is appended to the head flit immediately before traversing the crossbar switch. Finally, flits are registered at the output modules and then traverse inter-router links.

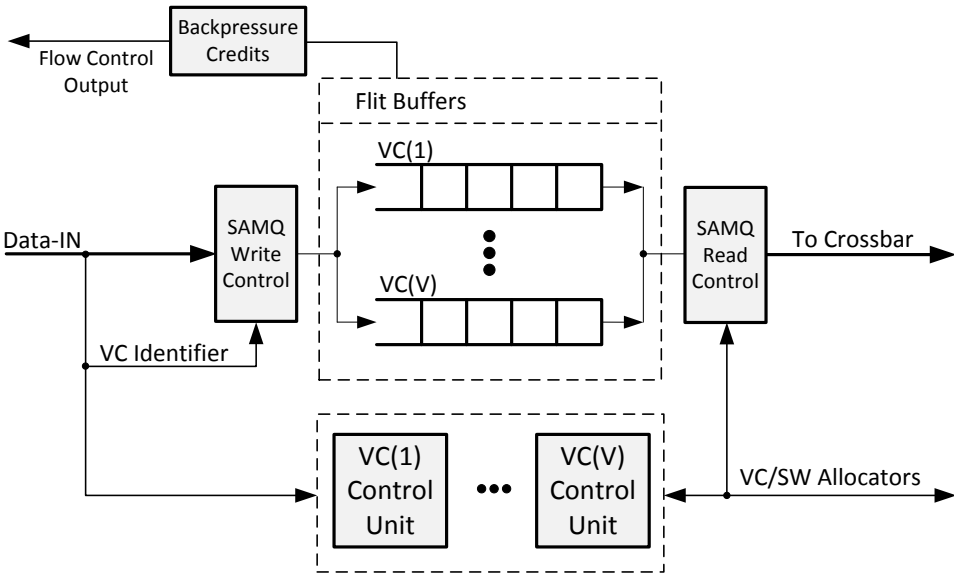


Figure 3.2: Input module for one router port and "V" virtual channels. Not all connections are shown.

Figure 3.1 shows a block diagram of the router. For each of the presented router components there are different implementation variants. We restrict our discussion to architectures that satisfy current NoC cost limitations and bandwidth demands; that is, current implementation norms. For further reading on NoC microarchitecture, the following references provide an in-depth discussion of implementation variations for each component [27, 52, 59, 112, 139].

3.1.1 Input Module

The main function of the input module is to buffer incoming flits until routing and resource allocation are complete. Depending on the VC identifier of the packet, it is stored in a different part of the input buffer. Routing information, already computed by the preceding router hop, is decoded and forwarded to the VC and switch allocators to bid for VCs and switching resources. The flit remains in the buffer until both a VC is allocated and the switch is free for traversal, at which point route lookahead information is attached to the head flit and it is ejected from the input module onto the switch.

Since route computation is done one hop earlier and in parallel to switch allocation, it does not impede router latency and effectively removes this step from the router pipeline [112]. Moreover, the low-overhead route computation logic is replicated for each VC to compute the route for all input ports simultaneously and support the queuing of multiple packets per VC [25]. A two-phase routing algorithm, known as Valiant's, routes first to a random intermediate node then to the destination node to improve load balancing [143]. The algorithm used to route each of the two phases is dimension-ordered routing which routes in each dimension sequentially and deterministically [52].

Dual-ported memory implements the input buffer. Internally, it is organized as a statically allocated multi-queue (SAMQ) buffer which divides the memory into equal portions for each VC [139]. Memory width is always the same as flit width to allow reading and writing flits in one cycle [25]. In this implementation, the memory buffer has one write port and two read ports to allow the queuing of more than one packet in a VC buffer. This is required for simultaneous loading of a packet's tail flit and the

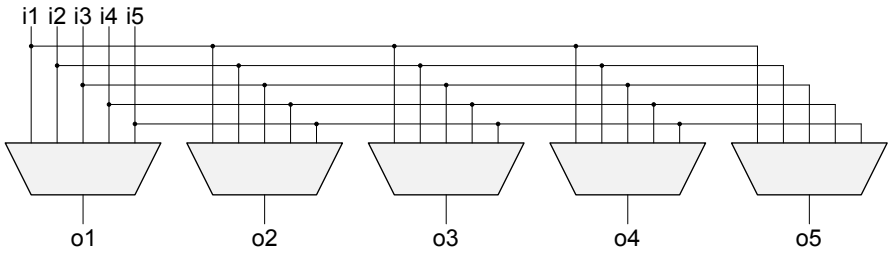


Figure 3.3: A 5-port multiplexer-based crossbar switch.

next packet's head flit to update the destination port for the new route. Without this optimization, a pipeline stall is introduced between packets.

Figure 3.2 shows a block diagram of the input module used in this study. The VC control units include the route-computation logic and state registers to keep track of the input VC status, the destination output port and the assigned output VC [25]. The SAMQ controller governs the read and write ports of the flit buffer. It selects the write and read addresses depending on the input VC and the granted output VC from switch allocation respectively. A backpressure control unit tracks buffer space availability per VC and transmits credits to upstream router ports on dedicated flow control links.

3.1.2 Crossbar

The crossbar is multiplexer-based as depicted in Figure 3.3, rather than a tri-state buffer crossbar. Moderns FPGAs can only implement crossbars with multiplexers, and modern ASIC crossbars are also usually implemented this way. In the best-case scenario, five flits may traverse the crossbar simultaneously if each of them is destined for a different output port. However this rarely occurs, thus requiring the queuing of flits in the input module buffers until the output port is free and the flit can proceed. We found the ASIC crossbar area to be gate limited and not wire limited for the range of parameters considered here; other recent work has shown that crossbars as large as 128 ports are also gate limited [118].

3.1.3 Virtual Channel Allocator

VC allocation is performed at packet granularity. Like route computation, the body flits inherit the decision made for the head flit. The route computation unit selects an output port for a packet, and any available VC on that output port is a candidate VC for the packet.

Figure 3.4 shows the separable input-first VC allocator used in our study. The first stage filters requests for output VCs by selecting a single output VC request for each input VC. This ensures that each input VC will be allocated a maximum of one output VC, which is necessary for correctness. The second stage ensures that each output VC is not over-allocated. For each output VC, where P is the number of ports and V is the number of VCs per port, the second stage takes $P \times V$ requests from all input VCs and grants one request. A virtual connection is established from the granted input VC to the output VC for the duration of one packet.

Arbiters grant requests in a round robin fashion which ensures that the most-recently granted requester has the lowest priority in the next round of arbitration [52]. For a large number of inputs, arbiters

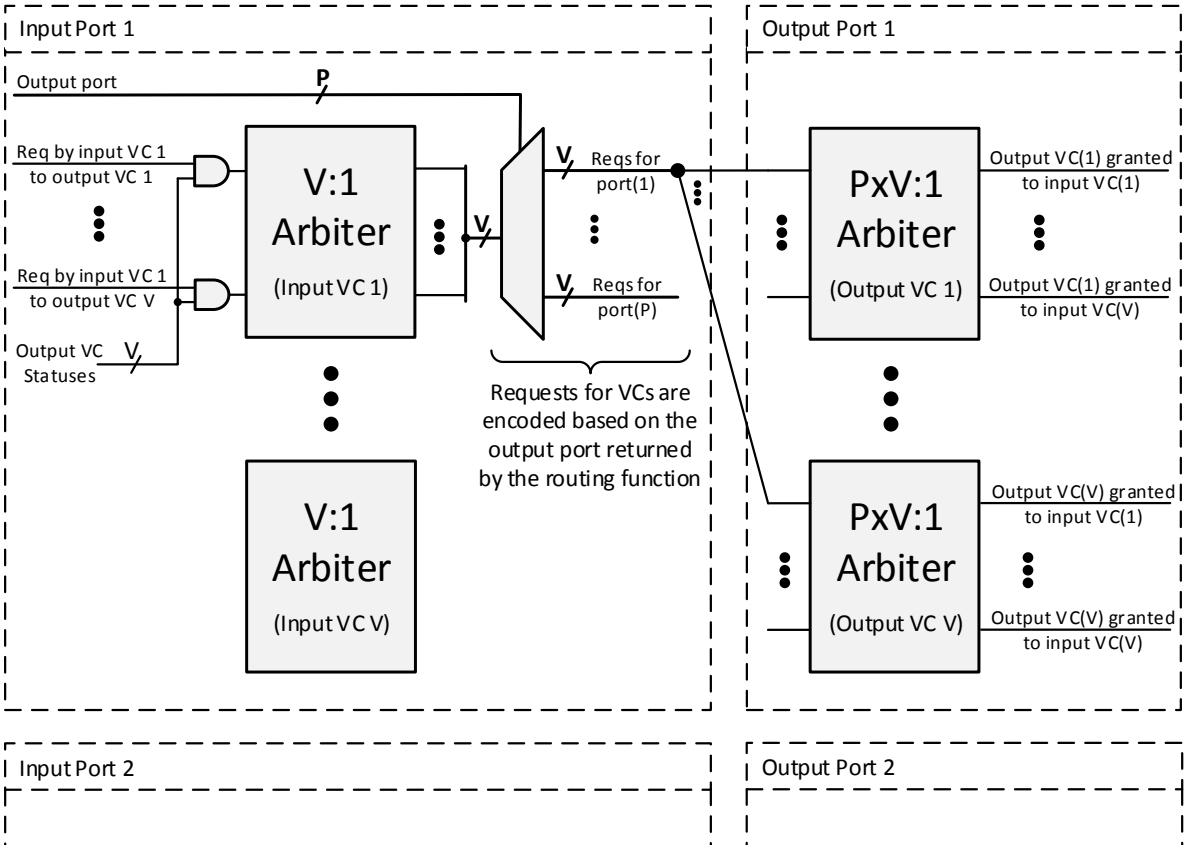


Figure 3.4: Separable input-first VC allocator with "V" virtual channels and "P" input/output ports.

have long combinational delays. To circumvent this limitation, allocators are organized hierarchically as tree arbiters, trading area for delay [27].

3.1.4 Switch Allocator

A switch allocator matches requests from $P \times V$ input VCs to P crossbar ports. Unlike routing and VC allocation, this is done on the flit granularity in a wormhole-switched router [52]. Figure 3.5 shows a separable input-first switch allocator. In the first stage of switch allocation, VCs in a single input port compete among themselves for a crossbar input port. The granted VC forwards its requests to the second stage of arbitration which selects between all input ports bidding for each of the output ports. This ensures that only one VC can attempt to access the crossbar from each input port, and that only one crossbar input can connect to a crossbar output at a given time.

Speculative switch allocation can reduce the router latency from 3 cycles to 2 by performing VC and switch allocation together, provided that a VC is allocated in this cycle [119]. Architecturally, this duplicates the switch allocator; one instance handles non-speculative requests and the other handles speculative ones. Priority is given to non-speculative requests using reduction logic and selection circuitry [27].

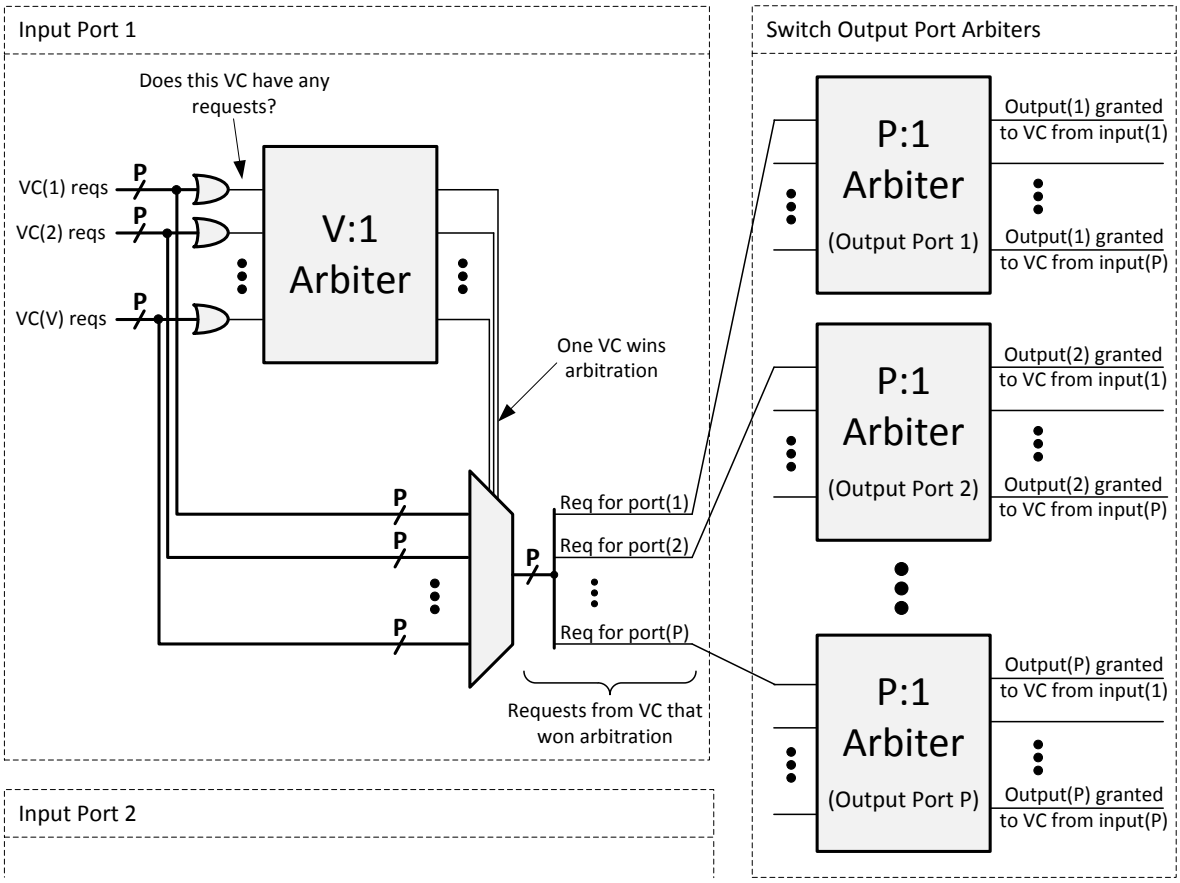


Figure 3.5: Separable input-first switch allocator with "V" virtual channels and "P" input/output ports.

3.1.5 Output Module

The crossbar output can be connected to the outgoing wires and the downstream routers directly. However, to improve clock frequency a pipeline stage is placed at the crossbar outputs [25]. Furthermore, the output registers are replicated per VC to buffer an additional flit before proceeding downstream allowing one extra flit to traverse the crossbar before receiving credits from the downstream router.

3.2 Links

Routers are connected to each other through unidirectional wires from a router output port, to its downstream router input port. In some NoCs, these links can be pipelined to improve their frequency [52], but we do not opt for that option because the router typically contains the critical path delay for the parameters we investigate [2, 5].

In ASIC implementations, these point-to-point links are simply implemented as metal wires in the ASIC chip metal stack. However, on FPGAs connections are made using the FPGA's programmable interconnect which contains different-size wire segments and multiplexers between any two points on the FPGA. We investigate both ASIC and FPGA links as two implementation candidates for NoC links. While soft FPGA links are more configurable, they are expected to be less efficient compared to hard ASIC links – we explore this tradeoff in Chapter 5.

Chapter 4

Methodology

Contents

Our goal is to study embedded NoCs on FPGAs. The first step is to compare embedded hard NoCs to soft NoCs built from the exisiting FPGA fabric. For each NoC subcomponent, we investigate the hard (ASIC) and soft (FPGA) implementation to inform the decision of whether we should embed any of the NoC components in hard logic. To that end, we quantify the area, speed and power advantages of hardening each of the NoC components. This chapter presents the methodology for measuring these efficiency metrics on both the ASIC (hard implementation) and FPGA (soft implementation) platforms. Hard versus soft comparisons have been performed previously on a range of different circuits [89]. We follow the methodology of that previous work, and the common practices used by digital designers to find realistic area and delay estimates on the ASIC and FPGA CAD flows. Finally, we verify our methodology against the area and frequency gap of a benchmark circuit used in previous work as an additional sanity check on our methodology [89].

4.1 Routers

We perform a detailed component-level analysis of the NoC by breaking down the router into its subcomponents and measuring the area, delay and power. We also vary the four main NoC parameters to understand how efficiency and performance of the NoC changes with these parameters. These parameters are width, number of ports per router, number of VCs, and buffer depth in the router input ports. We list the baseline values of these parameters, and their ranges in Table 4.1. We choose parameters that correspond to reasonable implementations of NoCs in previous work [52].

Table 4.1: Baseline and range of NoC parameters for our experiments.

	Width	Num. of Ports	Num. of VCs	Buffer Depth
Baseline	32	5	2	5/VC (10)
Range	16-256	2–15	1–10	5–65 /VC

The router is implemented both on the largest Altera Stratix III FPGA (EP3SL340) and the 65 nm ASIC process technology from Taiwan Semiconductor Manufacturing Company (TSMC). This allows a direct FPGA vs. ASIC comparison since Stratix III devices are manufactured in the same 65 nm TSMC process technology [15]. Moreover, the area for Stratix III resources is publicly available which allows a direct head-to-head comparison [145].

4.1.1 FPGA CAD Flow

Table 4.2 shows the area, including interconnect, of the various FPGA blocks. We use these block areas to find the equivalent silicon area of NoC components implemented using soft logic on an FPGA.

Table 4.2: Estimated FPGA Resource Usage Area [145]

Resource	Relative Area (LAB)	Tile Area (mm^2)
LAB	1	0.0221
ALM	0.10	0.0022
ALUT (half-ALM)	0.05	0.0011
BRAM - 9 kbit	2.87	0.0635
BRAM - 144 kbit	26.7	0.5897
DSP Block	11.9	0.2623

To implement router components on the FPGA, we use Altera Quartus II v11.1 software with the highest optimization options. This excludes physical synthesis which reduced critical path delay by 5% and increased area by 15% on the router components that we analyze. We set an impossible timing constraint of 1 GHz to force the tools to optimize for timing aggressively and report the maximum achievable frequency. While 1 GHz is clearly not a realizable constraint, we found it achieved timing and area optimization results comparable to those obtained with a timing constraint that is difficult, but just barely achievable. Clock jitter and on-die variation are modeled using the "derive_clock_uncertainty" command which applies clock uncertainty constraints based on knowledge of the clock tree [132].

All the circuit I/Os, except the clock, are tied to LUTs using the "virtual pin" option. This mimics the actual placement of an NoC router, and avoids any placement, routing or timing analysis bias that could result from using actual FPGA I/O pins. For example, the placement of the NoC component could be highly distorted by a large number of connections to I/Os located around the device periphery, when in a real system the input and outputs of the component would be within the fabric. Additionally, the use of virtual pins allows the compilation of designs with more I/Os than physically available on the FPGA.

Resource utilization is used to calculate the occupied FPGA silicon area by multiplying the used resource count by its physical area in Table 4.2. Simply counting the used logic array blocks (LABs) or adaptive logic modules (ALMs) can overestimate the area required, as many LABs and ALMs are only

partially occupied, and could accept more logic in a very full design. Instead, we use the post-routing area utilization from the resource section in the fitter report which accounts for the porosity in packing, thereby giving a realistic area estimate for a highly utilized FPGA. Note that the LUTs used for "virtual pins" are subtracted out.

The fastest FPGA speed grade for Stratix III devices corresponds to typical transistors, whereas the slowest FPGA speed grade matches the worst-case transistors of a process. For the best comparison, we use the fastest FPGA speed grade and the typical transistor model for the ASIC tools. For purely combinational modules, such as the crossbar, registers are placed on the inputs and outputs to force timing analysis. Maximum delay is extracted from the timing reports using the most pessimistic (slow, 85 ^{o}C) timing model, assuming a 1.1 V power supply.

4.1.2 ASIC CAD Flow

Synopsys Design Compiler vF-2011.09-SP4 is used for synthesis, and area and delay estimation. The general-purpose typical process library is used with standard threshold voltage and 0.9 V supply voltage. Unlike the FPGA CAD flow, timing constraints and optimizations impact the ASIC area dramatically, as ASIC timing optimizations entail standard cell upsizing and buffer insertion whereas FPGA subcircuits are fixed. For this reason, a two-step compilation procedure, described below, is used to reach a realistic point in the large tradeoff space between area and delay.

We perform compilation using a top-down flow, with "Ultra-effort" optimizations for both area and delay. This turns on all optimization options in the synthesis algorithm and accurately predicts post-layout critical path delay and area using topographical technology [138]. All registers are replaced with their scan-enabled equivalent to allow the necessary post-manufacturing testing for ASICs. We use a conservative wire model from TSMC in which the capacitance and resistance per unit wire length are used together with a fanout-dependent length model to estimate the wire delay. Additionally, these parameters are automatically adjusted based on the area of the design hierarchy spanned by each net.

In step one, we perform an ultra-effort compilation with an impossible 0 ns timing constraint and extract the negative slack of the critical path from the timing report. Area numbers are bloated when trying to satisfy the impossible timing constraints and are discarded from this compilation. The negative slack from step one is used as the target clock period in step two of the ASIC compilation. This provides a reasonable target for the CAD tools and results in realistic cell upsizing, logic duplication and buffer insertion, and hence realistic area numbers. With the clock period adjusted, the design is recompiled and the implementation area and delay are extracted from the synthesis reports. Note that any positive or negative slack in this step is also added to the critical path delay measurement.

Generally ASICs are not routable if they are 100% filled with logic cells. To account for whitespace, buffers inserted during placement and routing, and wiring, we assume a 60% rule-of-thumb fill factor and so inflate the area results by 66.7%. Fill factors as low as 10% and as high as 90% have been used in the literature [89, 118] but we chose 60% to model the typical case after conversations with ASIC design engineers.

4.1.3 Power Simulation

We generate the post-layout gate-level netlist from the FPGA CAD tools (Altera Quartus II v11.1) and the post-synthesis gate-level netlist from the ASIC CAD tools (Synopsys Design Compiler vF-2011.09-

SP4) as outlined above. For accurate dynamic power estimation, we first simulate these gate-level netlists with a testbench to extract realistic toggle rates for each synthesized block in the netlists.

The testbench consists of data packet generators connected to all router inputs and flit sinks at each router output. The packet generator understands back pressure signals from the router, so it stops sending flits if the input buffer is full. We attempt to inject random flits every cycle into all inputs and we accept flits every cycle from outputs to maximize data contention in the router, thus modeling an upper bound of router power operating under worst-case synthetic traffic. We perform a timing simulation of the router in Modelsim for 10000 cycles and record the resulting signal switching activity in a value change dump (VCD) file. Note that we disregard the first and last 200 cycles in the testbench so that we are only recording the toggle rates for the router at steady state and excluding the warm-up and cool-down periods.

This simulation is very accurate for two main reasons. First, by simulating the gate-level netlist we obtain an individual toggle rate for each implemented circuit block. Second, we perform a timing simulation that takes all the delays of logic and interconnect into account; consequently the toggle rates are highly accurate and include realistic glitching. It is then a simple task for power analysis tools to measure the power of each synthesized block (LUTs, interconnect multiplexers or standard cells) by using their power-aware libraries and the simulated toggle rates on each block input and output.

We use the extracted toggle rates to simulate dynamic power consumption, per router component, for both the FPGA and ASIC implementations using their respective design tools: Altera's PowerPlay Power Analyzer for the FPGA and Synopsys Power Compiler for the ASIC. The nominal supply voltage for the TSMC 65 nm technology library is 0.9 V compared to 1.1 V for the Stratix III FPGA. For that reason, we scale the ASIC dynamic power quadratically (by multiplying by $\frac{1.1^2}{0.9^2}$) when computing FPGA-to-ASIC power ratios. In all other power results, we explicitly state which voltage we are using.

4.1.4 Methodology Verification

To verify the methodology, we compare the results obtained with our methodology to those of Kuon and Rose on their largest benchmark, raytracer [89]. As shown in Table 4.3, the area and delay ratios are quite close; we expect some difference as our results are from a 65 nm process while theirs are from 90 nm.

Table 4.3: Raytracer area and delay ratios.

	Kuon and Rose [89]	This Work
FPGA Device	Stratix II	Stratix III
ASIC Technology	90 nm	65 nm
Area Ratio	26.0	25.6
Delay Ratio	3.5	4.1

Note that we do not verify power ratios against previous work. This is mainly because we do not have access to the simulation testbench vectors which are necessary to obtain accurate toggle rates for the Raytracer benchmark. Furthermore, we believe that area and delay ratios are enough to improve confidence in our methodology, since power and area are typically highly correlated [89].

4.2 Links

NoC links consist of wires between router input ports and output ports. The two parameters that affect links are data width and distance between routers. We investigate wires on both the FPGA and ASIC platforms to quantify their efficiency and performance. In this section, we outline the methodology we followed in designing and evaluating these wires.

4.2.1 FPGA CAD Flow

Soft NoC links are implemented using the prefabricated FPGA "soft" interconnect. On Stratix III FPGAs, there are four wire types: vertical length four (C4) and length 12 (C12), and horizontal length four (R4) and length 20 (R20). We connect two registers using a single wire segment to measure the delay and dynamic power of this wire segment. Next, we investigate different connection lengths by connecting wire segments of the same type in series and measuring delay and power. Registers are manually placed using location constraints to define the wire endpoints, and the connection between the registers is manually routed by specifying exactly which wires are used in a routing constraints file (RCF).

Wire delay is measured using the most pessimistic (slow, 85 ^{o}C) timing model. The dynamic power consumed by the wires is linearly proportional to the toggle rate. 0% means that the wire has a constant value, while 100% means data toggles on each positive clock edge. For each simulated router instance, we extract the toggle rates at its inputs and outputs and use that to simulate the wire power. This ensures that the data toggle rates on the NoC links correctly match the router inputs and outputs to which the links are connected.

4.2.2 ASIC CAD Flow

We use TSMC's metal properties to simulate lumped element models of wires allowing us to measure the delay and power of ASIC NoC links. Metal resistance and capacitance are provided with the TSMC 65 nm technology library for each possible wire width and spacing on each metal layer. Metal layers are divided into three groups based on the metal thickness: local, intermediate and global.

1. Local wires: Lowest metal layer used to connect transistors.
2. Intermediate wires: The middle 6 metal layers, used to construct buses and other short/long connections on chip.
3. Global wires: The fastest, widest wires on-chip are available on the top 2 metal layers and are used for clock networks or other fast chip-wide connections.

In our measurements, we use the intermediate wires because, unlike the alternatives, they are both abundant and reasonably fast. We calculate the resistance and capacitance of a wire using intermediate-metal-layer metal data from TSMC's 65 nm technology library. We use Synopsys HSPICE vF-2011.09.SP1 to simulate a lumped element (π) model of hard wires [123]. Propagation delay is measured for both rising and falling edges of a square pulse signal, and the worst case is taken to represent the speed of this wire. Dynamic power is computed using the equation ($P = \frac{1}{T}\int_0^T V\, I(t)\, \mathrm{d}t$) and it is scaled linearly to the routers' toggle rates.

We design and optimize the ASIC interconnect wires to reach reasonably low delay and power comparable to FPGA wires by choosing:

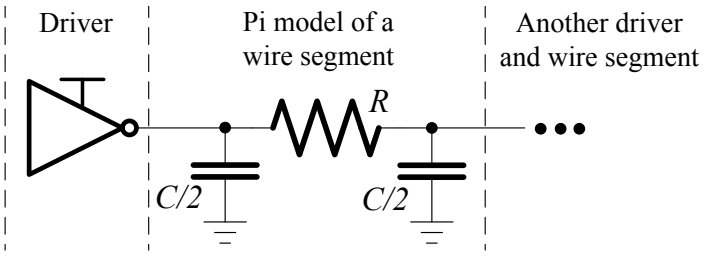


Figure 4.1: RC π wire model of a wire with rebuffering drivers.

1. Wire width and spacing: This determines the parasitic capacitance and resistance in a wire segment (C and R in Figure 4.1) which determines its delay and power dissipation.
2. Drive strength: The channel width of transistors used in the interconnect driver. Affects speed and power.
3. Rebuffering: How often drivers are placed on a long wire.

Using the π wire model, we conducted a series of experiments using HSPICE to optimize our ASIC wire design. To match the FPGA experiments, the supply voltage was set to 1.1 V and the simulation temperature at 85 ^{o}C. We also repeated our analysis at 0.9 V for the low-power version of our hard NoC. We reached a reasonable design point with metal width and spacing of 0.6 μm, drive strength of 20-80$\times$ that of a minimum-width transistor (depending on total wire length) and rebuffering every 3 mm. If necessary, faster or lower power ASIC wires could be designed with further optimization or by using low-swing signaling techniques [53].

Chapter 5

NoC Component Analysis

Contents

In this chapter we use the router microarchitecture from Chapter 3, and the methodology in Chapter 4 to perform a component-level analysis of NoC components. We measure and compare the area, speed and power of routers (divided into 5 subcomponents) and links. We quantify the efficiency and performance gaps when implemented hard and soft to better understand the two options and to explore the design space of implementing NoCs on FPGAs.

5.1 Routers

We analyze the area, delay and power ratios of each router subcomponent in this section. We also combine our results to find overall router efficiency and performance differences between soft and hard implementations.

5.1.1 Area and Speed

Figures 5.1 and 5.2 show the FPGA/ASIC area and delay ratios for the router components as they vary with the four main router parameters: flit width (16-256), number of ports (2-15), number of VCs (1-10) and input buffer depth (5-65) as shown in Table 4.1. These results are summarized in Table 5.1 in which the minimum, maximum and geometric mean is given for each component. Additionally, we give the results for a complete VC router built out of those components. We choose a realistic range of router parameters, based on a study of the literature, such that the geometric average of the area or delay ratio is indicative of the FPGA-to-ASIC gap for an NoC that is likely to be constructed. On

average, NoC routers use 30× less area and run 6× faster when embedded in hard logic compared to a soft implementation (see Table 5.1).

Table 5.1: Summary of FPGA/ASIC (soft/hard) router area and delay ratios.

Router Component	**Area**			**Delay**		
	Min.	Max.	**Geomean**	Min.	Max.	**Geomean**
Input Module	8	36	**17**	2.2	4.0	**2.9**
Crossbar	57	169	**85**	3.3	6.9	**4.4**
VC Allocator	27	76	**48**	2.0	4.8	**3.9**
Switch Allocator	24	94	**56**	1.9	4.2	**3.3**
Output Module	30	47	**39**	3.1	3.7	**3.4**
Router	13	64	**30**	4.7	8.0	**6.0**

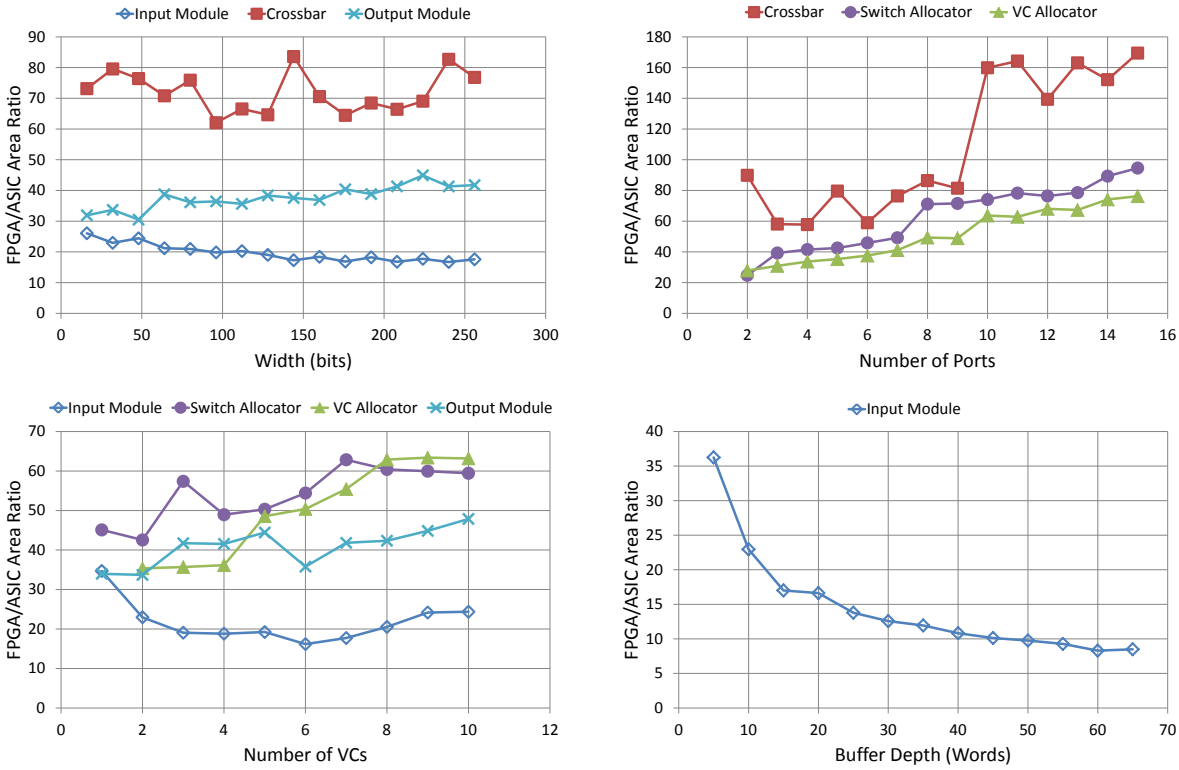


Figure 5.1: FPGA/ASIC (soft/hard) area ratios as a function of key router parameters.

Input Module

The input module consists of a memory buffer and logic for routing and control. To synthesize an efficient FPGA implementation, the memory buffer is modified to target the three variants of RAM on FPGAs: registers, lookup table random access memory (LUTRAM)[1] and BRAM. LUTRAM uses FPGA LUTs as small memory buffers and BRAMs are dedicated hard memory blocks that support tens

[1]Stratix IV was used for LUTRAM experiments to avoid a bug in Stratix III LUTRAM.

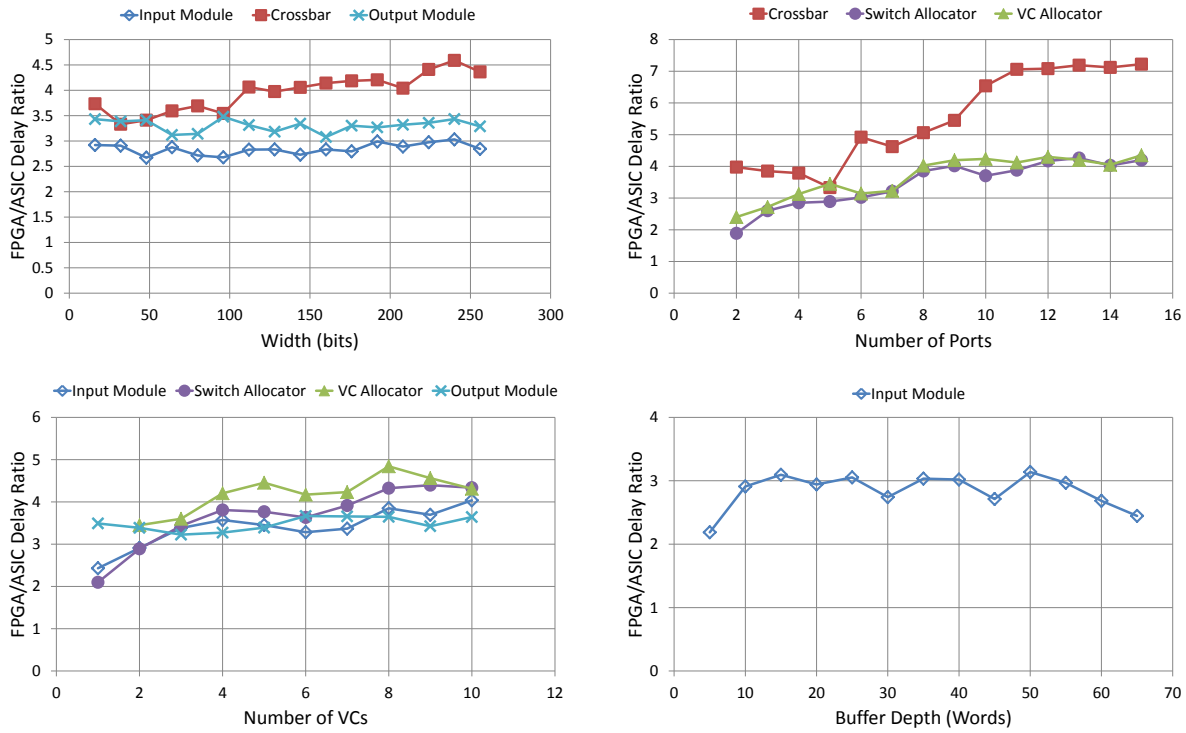


Figure 5.2: FPGA/ASIC (soft/hard) delay ratios as a function of key router parameters.

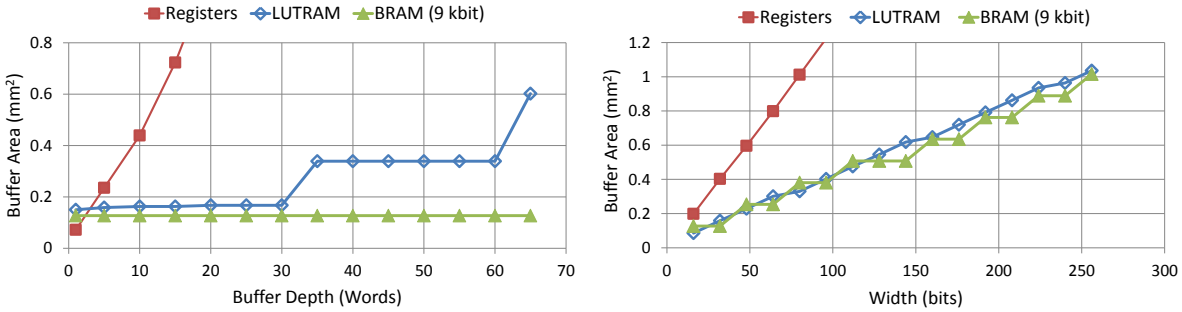


Figure 5.3: FPGA silicon area of memory buffers implemented using three alternatives.

of kilobits. Both LUTRAM and BRAM can implement dual-ported memories (1r1w) and the second read port (2r1w) required for the input module is added by replicating the RAM module. Registers are more flexible and can construct multiple read ports by replicating the read port itself and not the storage registers. On ASICs, the memory buffer is always implemented using a 2D flip-flop array which is the norm for building small memories.

Figure 5.3 shows the FPGA area of various buffers when implemented using the three alternatives mentioned. The minimum-area implementation is selected for the comparison against ASICs. In all the data points when varying the buffer depth, the BRAM-based implementation has the lowest area. In fact, the area remains constant since the 9-kbit BRAM can handle up to 256 memory words. LUTRAM is slightly less efficient than BRAM with shallow buffers, but the area increases rapidly whenever another LUTRAM is used to increase buffer depth. BRAMs and LUTRAMs have width limitations but can be

grouped together to implement wider memories. This explains the linear area increase with width shown in Figure 5.3.

When the memory buffers are built out of registers, there are no bits wasted. LUTRAM can only be instantiated in quantized steps of the LUTRAM size (640 bits); hence some bits can go unused. This is more true for BRAM which can only be instantiated in steps of 9 kbits for the Stratix III architecture. Nevertheless, the bit density for a register-based memory buffer is 0.77 kbit/mm^2 compared to 23 kbit/mm^2 for a LUTRAM and 142 kbit/mm^2 for a 9-kbit BRAM [145]. This means that a 9-kbit BRAM with only 16% of its bits used is just as area-efficient as a fully utilized LUTRAM on a Stratix III FPGA, explaining the lower BRAM area with very shallow buffers. Although prior work has gravitated towards the use of LUTRAM [117], when looking at it from a silicon perspective, the high density of BRAM makes it more area-efficient for most width×depth combinations. However, in architectures with deeper BRAM, such as Virtex 7, LUTRAM may be the more efficient alternative for shallow buffers.

As Table 5.1 shows, the input module has the lowest area and delay gaps of the presented components. The area gap varies from 8-36× with a geometric mean of 17×. The lower gap occurs when a deep buffer is used and the FPGA BRAM is well-utilized. Width has only a small effect on the input module area and delay ratios, but varying the number of VCs presents a more interesting result. The input module consists of both control logic, which is inefficient on FPGAs, and memory buffers implemented as compact hard blocks. As we vary the number of VCs in Figure 5.1 the FPGA implementation becomes twice as efficient between 1 and 6 VCs because we are able to pack more buffer space into the same BRAM module. However, as we increase the number of VCs further, the efficiency drops because the control logic for a large number of VCs becomes the dominant area component. The FPGA-to-ASIC delay ratio is 2.9× and is always limited by the logic component of the input module and not the fast BRAM memory component.

Crossbar

Crossbars show the largest area gap; a minimum of 57×, a maximum of 169× and a geometric average of 85×. It is worth noting that there is a 2× FPGA efficiency loss for crossbars with 10 or more ports. This is due to two causes. First, the required FPGA LUTs per multiplexer port increases faster than the ASIC gates per port. Second, there is an increased demand for interconnect ports at the logic module inputs causing LUTs in that logic module to be unusable; the ratio of LUTs that are unused for this reason grows from 1% to 10% between 9 and 10 multiplexer ports. When the width is varied however we see very little variation between a 16-bit and a 256-bit wide crossbar; the variations in the width plot are due to better or worse mapping of different-size multiplexers onto the FPGA's LUTs.

The crossbar delay gap grows significantly from 3.3-6.9× with increasing port count. This trend is due to the increase in FPGA area, which causes the multiplexers to be fragmented over multiple logic modules thus extending the critical path. Overall, the average delay gap is 4.4× for the crossbar; the largest out of all the components.

The results show that crossbars are inefficient on FPGAs and their scaling behavior is also much worse than ASICs. This is a prime example of a circuit that would bring area and delay advantages if it were hardened on the FPGA. In this scenario, the crossbar can be overprovisioned with a large number of ports so that it can support different NoC organizations. If a small number of router ports are required but a large number of ports are available, the additional ports can be used towards crossbar

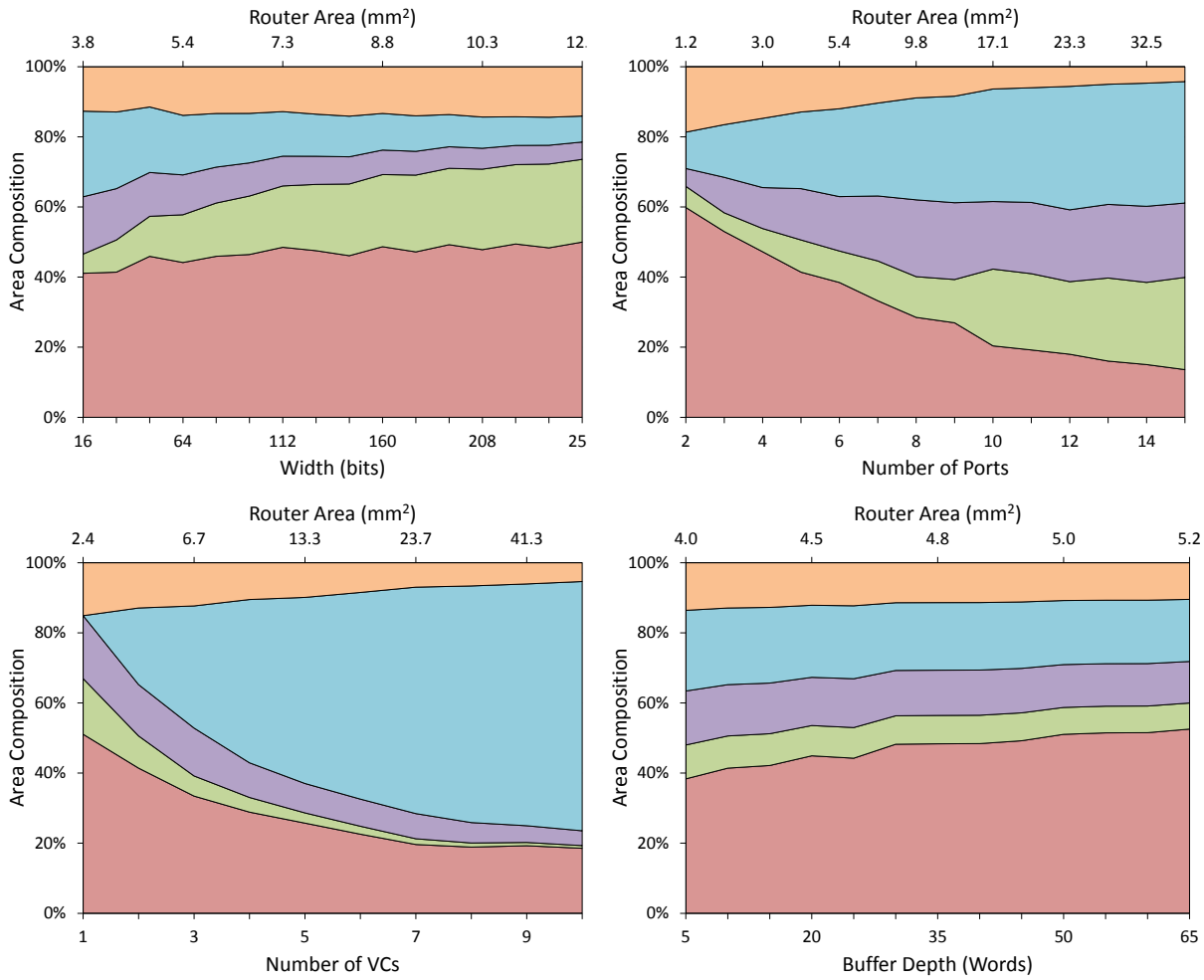


Figure 5.4: FPGA (soft) router area composition by component. Starting from the bottom(red): Input module, crossbar, switch allocator, VC allocator and output module.

speedup which relieves crossbar traffic by allowing multiple VCs from the same input port to traverse the switch simultaneously. This also simplifies switch allocation [52].

VC and Switch Allocators

Allocators are built out of arbiters which consist of combinational logic and some registers. Ideally the ratio of LUTs to registers should match the FPGA architecture; for Stratix III a 1:1 ratio would use the resources most efficiently. Deviation from this ratio means that some logic blocks will have either registers or LUTs used but not both. The unused part of the logic block is area overhead when compared to ASICs.

Although there are other sources of FPGA inefficiencies, there is a direct correlation between the LUT-to-register ratio and the FPGA-to-ASIC area gap. For the VC allocator the average LUT-to-register ratio is 8:1 and the area gap is 48×, while the speculative switch allocator has an average LUT-to-register ratio of 20:1 and the area gap is higher; approximately 56×. This difference between

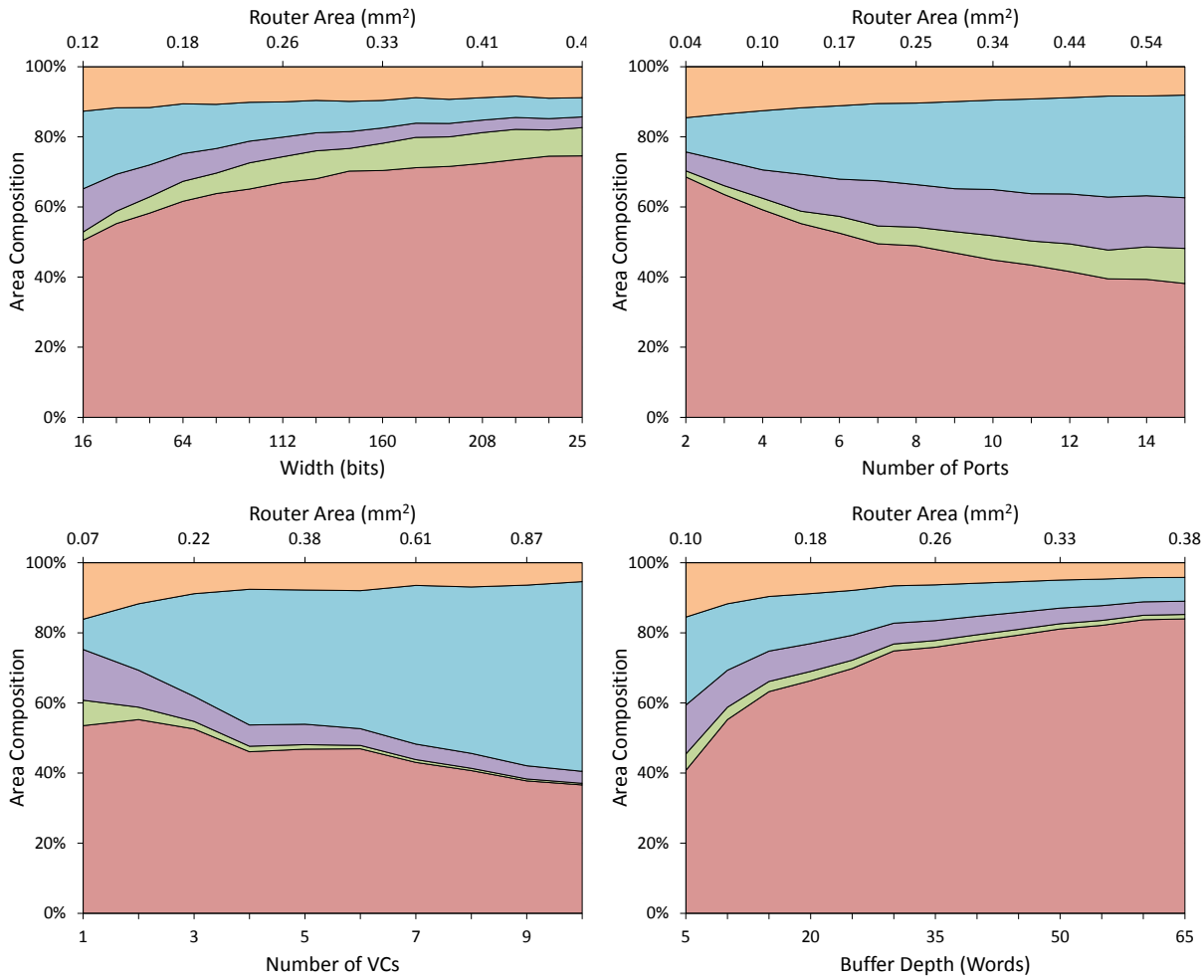


Figure 5.5: ASIC (hard) router area composition by component. Starting from the bottom(red): Input module, crossbar, switch allocator, VC allocator and output module.

the two allocators is due to the selection logic which is used in the speculative switch allocator and absent from the VC allocator.

Allocator delay increases with circuit size for both hard and soft implementations; however, the delay rises more rapidly for the soft version. Consequently, the delay ratio of the allocators is proportional to the circuit size, and grows with increasing port or VC count. We suspect this is because the fixed FPGA fabric restricts the placement and routing optimizations that can be performed on large circuits while ASIC flows have more options, such as upsizing cells or wires on critical paths. Overall, the delay gap is around 3.6× for the allocators.

Output Module

The output module is the smallest router component and is dominated by the output registers. Indeed, the LUT-to-register ratio is 0.6:1 contributing to its smaller area gap of 39× when compared to the allocators. The average delay ratio of 3.4× is also relatively low because the simple circuitry does not stress the FPGA interconnect.

Table 5.2: Summary of FPGA/ASIC power ratios.

Module	Min.	Max.	Geometric Mean
Input Module	3	23	10
Crossbar	15	194	64
Allocators	33	61	41
Output Module	14	19	16
Router	5	27	14

Router Area Composition on FPGA and ASIC

Figures 5.4 and 5.5 show the router area composition on FPGAs and ASICs respectively. Moreover, the total router area of select data points is given on the top axes.

The main discrepancy between the FPGA and ASIC router composition is the proportion of the input modules and the crossbar. The input modules are the largest components for most router variants on both the soft and hard implementations. It follows from the area ratios that the input modules are relatively larger on ASICs than on FPGAs; in fact, they occupy 36-83% of the ASIC router area compared to 14-60% on the FPGA. The crossbar is the smallest component of an ASIC VC router. On FPGAs, however, it becomes a critical component with a wide datapath or a large number of ports where it occupies up to 26% of the area.

With an increasing number of VCs, the VC allocator area dominates on both FPGAs and ASICs. Increasing the number of ports also increases the VC allocator area but to a lesser extent. This is due to the second stage of VC allocation which occupies most of the area and is constructed out of $P \times V$:1 arbiters. They require an additional P inputs per arbiter when the number of VCs is increased whereas only V additional inputs are required when the number of ports is raised. Since P is larger than V for the baseline router, the VC allocator's area grows more slowly with the number of ports than it does with the number of VCs. The speculative switch allocator also grows with increasing port and VC counts but is more affected by the number of ports. With 15 router ports, the switch allocator makes up 22% of the FPGA router area.

5.1.2 Dynamic Power

This section investigates the dynamic power of both hard and soft NoC components; only by understanding where power goes in various NoCs can we optimize it. Note that we do not investigate static power in this analysis. We divide the NoC into routers and links, and further divide the routers into four subcomponents. After sweeping four key design parameters (width, number of ports, number of virtual channels (VC) and buffer depth) we find the soft:hard power ratios for each router component as shown in Figure 5.6. We also investigate the percentage of power that is dissipated in each router component for both hard and soft implementations in Figures 5.7 and 5.8. Finally, we analyze the speed and power of NoC links whether they are constructed out of the FPGA's soft interconnect or dedicated hard (ASIC) wires.

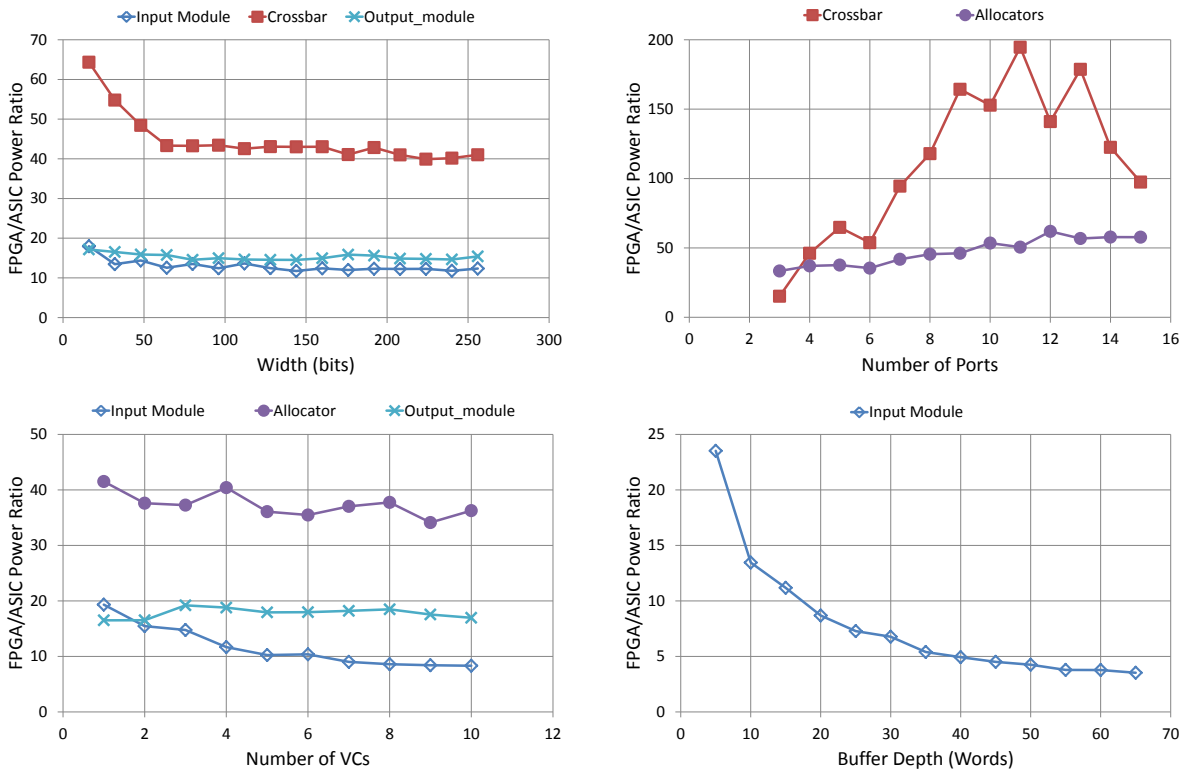


Figure 5.6: FPGA/ASIC (soft/hard) power ratios as a function of key router parameters.

Router Dynamic Power Ratios

As Table 5.2 shows, routers consume 14× less power when implemented hard compared to soft. When looking at the router components, the smallest power gap is 10× for input modules since they are implemented using efficient BRAMs on FPGAs. On the other hand, crossbars have the highest power gap (64×) between hard and soft. Note that there is a strong correlation between the FPGA:ASIC power ratios presented here and the previously presented NoC area ratios, while the power and delay ratios do not correlate well [2]. We believe this is because total area is a reasonable proxy for total capacitance, and charging and discharging capacitance is the dominant source of dynamic power.

Width: Figure 5.6 shows how the power gap between hard and soft routers varies with NoC parameters. The first plot shows that increasing the router's flit width reduces the gap. For example, 16 bit soft crossbars consume 65× more power than hard crossbars, while that gap drops to approximately 40× at widths higher than 64 bits. The same is true for input modules where the power gap drops from 18-12×. This indicates that the FPGA fabric is efficient in implementing wide components and encourages increasing flit width as a means to increase router bandwidth when implementing soft NoCs.

Number of Ports: Unlike width, increasing the number of router ports proved unfavorable for a soft router implementation. The allocators power gap is 57× at high port count compared to 35× at low port count. For crossbars, the power gap triples from 50× at six or less ports, to 150× with a higher number of ports. This suggests that low-radix soft NoC topologies, such as rings or meshes, are more efficient on traditional FPGAs than high-radix and concentrated topologies.

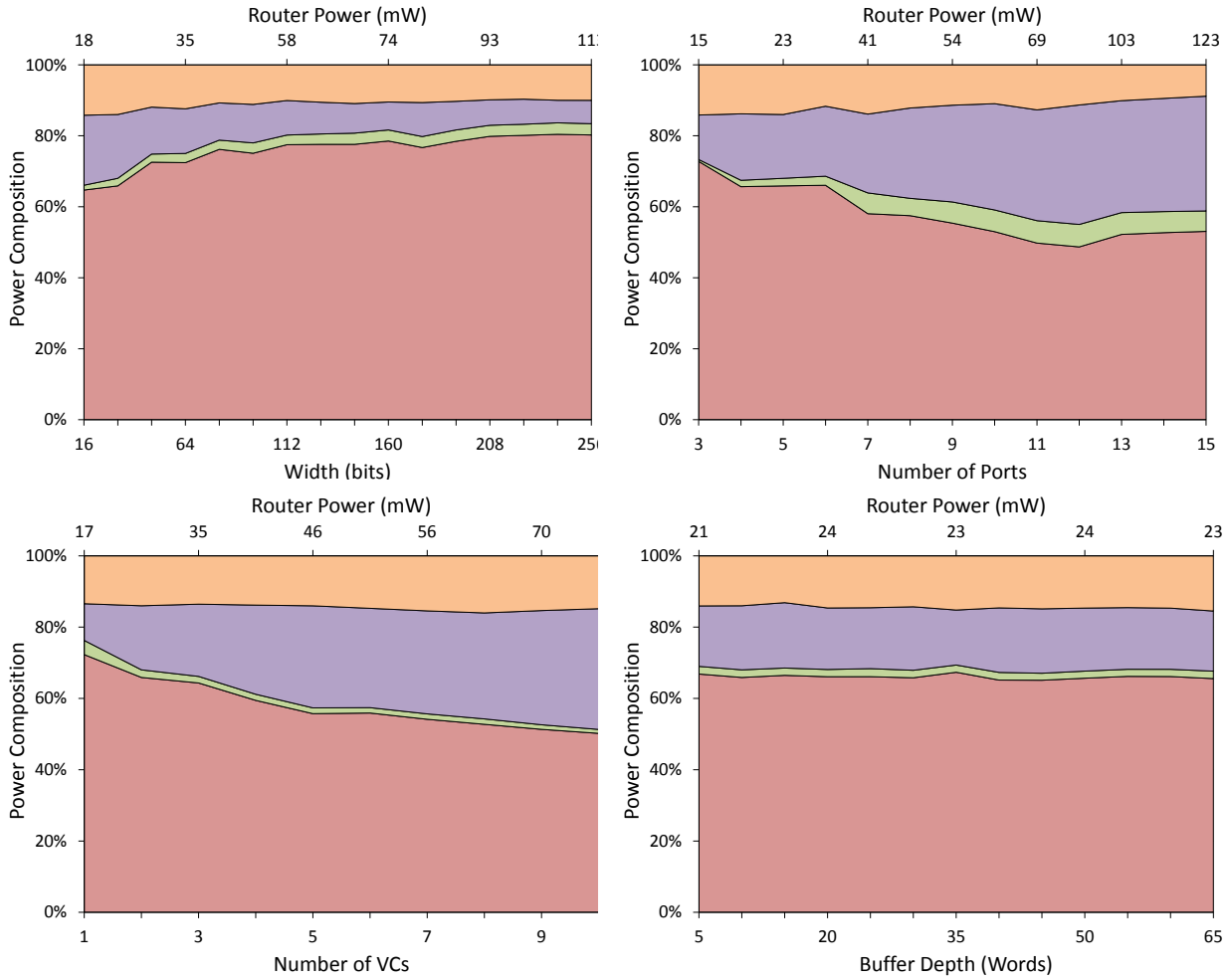


Figure 5.7: FPGA (soft) router power composition by component and total router power at 50 MHz. Starting from the bottom (red): Input modules, crossbar, allocators and output modules.

Number of VCs and Buffer Depth: Increasing the number of VCs is another means to enhance router bandwidth because VCs reduce head-of-line blocking [52]. This requires multiple virtual FIFOs in the input buffers and more complex control and allocation logic. Because we use BRAMs for the input module buffers on FPGAs, we have enough buffer depth to support multiple large VCs. Conversely, ASIC buffers are built out of registers and multiplexers and are tailored to fit the required buffer size exactly. As a result, the input module power gap consistently becomes smaller as we increase the use of buffers by increasing either VC count or buffer depth, as shown in Figure 5.6.

Allocators are composed of arbiters, which are entirely composed of logic gates and registers. Increasing the number of VCs increases both the number of arbiters and the width of each arbiter. The overall impact is a weak trend – the power ratio between soft and hard allocators narrows slightly as the number of virtual channels increases.

Router Power Composition

Figures 5.7 and 5.8 show the percentage of dynamic power consumed by each of the router components and the total router power is annotated on the top axes. Clearly most of the power is consumed by

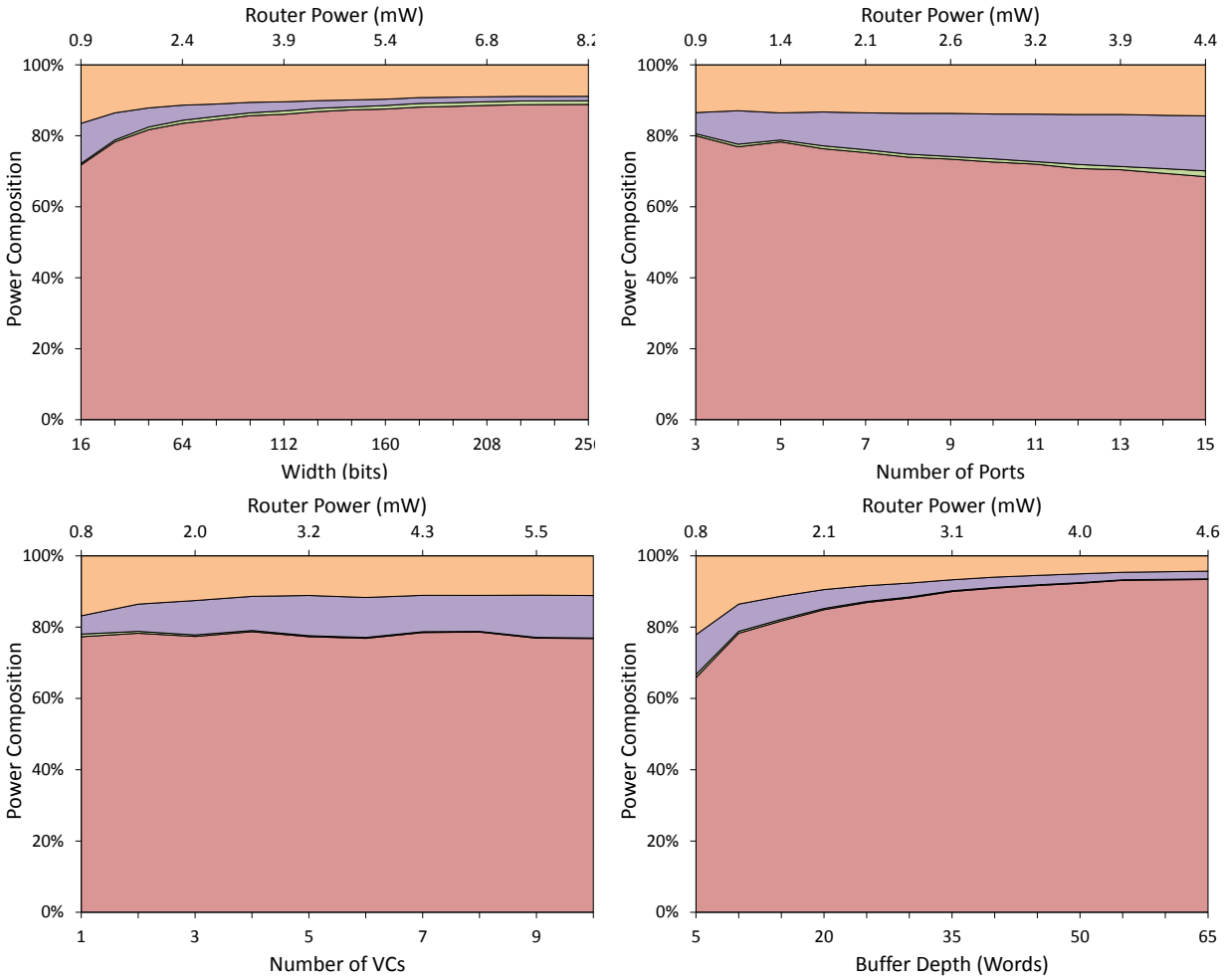


Figure 5.8: ASIC (hard) router power composition by component and total router power at 50 MHz. Starting from the bottom (red): Input modules, crossbar (very small), allocators and output modules.

the input modules, as shown by previous work [23, 70], but the effect is weaker in soft NoCs than in hard. This also conforms with the area composition of the routers; most of the router area is dedicated to buffering in the input modules, while the smallest router component is the crossbar [2]. Indeed, the crossbar power is very small compared to other router components as shown in the figures.

Next we look at the power consumption trends when varying the four router parameters. As we increase width, the router datapath consumes more power while the allocator's power remains constant. When increasing the number of ports or VCs, the proportion of power consumed by the allocators increases since there are more ports and VCs to arbitrate between. With deeper buffers, there is almost no change in the soft router's total power or its power composition. This follows from the fact that the same FPGA BRAM used to implement a 5-flit deep buffer is used for a 65-flit deep buffer. However, on ASICs there is a steady increase of total power with buffer depth because deeper buffers require building new flip-flops and larger address decoders.

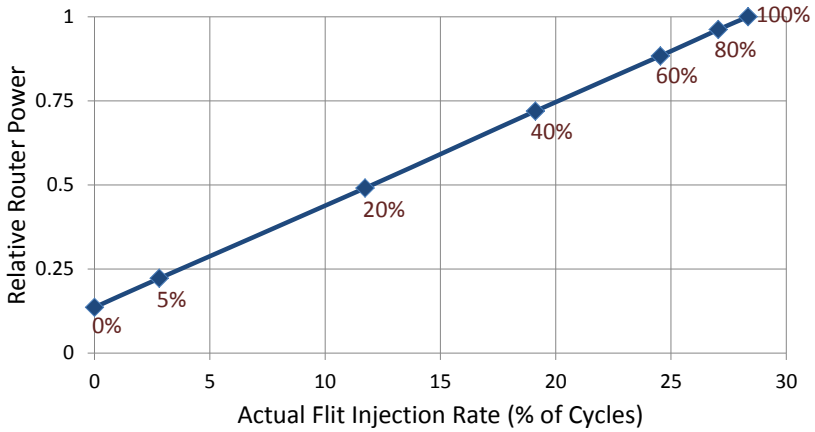


Figure 5.9: Baseline router power at actual data injection rates relative to the its power at maximum data injection. Attempted data injection is annotated on the plot.

Router Power as a Function of Data Injection Rate

Router power is not simply a function of area, it also depends very strongly on the amount of data traversing the router. A logical concern is that NoCs may dissipate more energy per unit of data under higher traffic. This stems from the fact that NoCs need to perform more (potentially power consuming) arbitration at higher contention levels, with no increase in data packets getting through. However, our measurements refute that belief. Figure 5.9 shows that router power is linear with the amount of data actually traversing the router, suggesting that higher congestion does not raise arbitration power. We annotate the attempted data injection rate on the plot. For example, 100% means that we attempt to inject data on all router ports on each cycle, but the x-axis shows that only 28% of the cycles carry new data into the router. This is because pseudo-random traffic from two router input ports will often be destined to the same output port causing one of the router inputs to stall. At zero data injection the router standby power, because of the clock toggling, is 13% of the power at maximum data injection, suggesting that clock gating the routers is a useful power optimization [111]. Importantly, router parameters also affect the data injection rate at each port.

- ***Width:*** Increasing port width does not affect the data injection rate because switch contention does not change. However, bandwidth increases linearly with width.
- ***Number of ports:*** Increasing the number of ports raises switch contention; thus the data injection rate at each port drops from 38% at 3 ports to 19% at 15 ports.
- ***Number of VCs:*** At 1 VC, data can be injected in 22% of the cycles and that increases to 32% at 4 VCs. Beyond 4 VCs, throughput saturates but multiple VCs can be used for assigning packet priorities and implementing quality of service guarantees [52].
- ***Buffer Depth:*** While deeper buffers increase the number of packets at each router, it does not affect the steady-state switch contention or the rate of data injection.

5.2 Links

Even though links are simply point-to-point connections between routers, they contain both a logic and metal component on both the FPGA and ASIC platforms. This is because a long wire implemented in

an ASIC chip requires rebuffering using transistor drivers to maintain a fast and readable signal with good integrity. On FPGAs, a point-to-point connection is built out of the programmable interconnect and consists of short wire segments, multiplexers and buffers – it therefore also contains a silicon and a metal component. In this section we discuss the area overhead (both silicon and metal), speed and power gaps between hard and soft wires.

5.2.1 Silicon Area

To find the transistor area overhead of soft wires, we look at the area required to implement programmable multiplexers at LAB inputs and outputs and assume that the type of multiplexers and interconnection flexibility used to connect routers to the programmable (soft) interconnect matches that of a LAB. 50% of a Stratix III LAB area (0.011 mm^2) consists of programmable interconnect and supplies 52 input ports and 40 output ports [98]. This means that each input or output occupies $\frac{0.011\ mm^2}{92} = 120\ \mu m^2$ of silicon area.

For hard wires, the area overhead is lower as only CMOS drivers are required at wire ends, and no multiplexers are needed. We use the exact transistor layout parameters from TSMC to hand layout the CMOS drivers and measure total transistor area. The drivers we use with 3 mm wires are 20× the minimum transistor width (=2.4 μm) which makes the total area of one CMOS driver 13 μm^2. Therefore hard wires are 9× more area efficient than soft wires, per router port, in terms of silicon area.

5.2.2 Metal Area

FPGAs use metal very heavily for interconnect; therefore it is a valuable and scarce resource, much like silicon, and must be studied to better understand the overhead of NoCs. Soft wires are spaced-out and often partially shielded to ensure glitch-free operation despite cross-talk for *any* signal combination that can be programmed onto them. The neighboring signals on hard links, on the other hand, are always other bits of the same link. This makes cross-talk modeling much simpler and can enable closer wire spacing and less shielding and thus increased metal efficiency over soft wires. However, it is difficult to obtain precise FPGA metal spacing values so we favour the FPGA platform and assume that hard and soft wires utilize metal equally for links that are the same length.

In Chapter 6 we compute the fraction of the FPGA or ASIC metal budget that is used for creating the NoC links. For FPGAs, we find the utilization of different soft interconnect resources (e.g. C4 and R4) and use that to estimate the metal area overhead of soft links. For hard links, we know the exact metal width (0.6 μm) and spacing (0.6 μm), and we use those to calculate the exact metal utilization.

5.2.3 Speed and Power

Figures. 5.10 and 5.11 shows the speed and power of hard and soft wires. Soft wires connect to multiplexers which increases their capacitive and resistive loading, making them slower and more power hungry. However, these multiplexers allow the soft interconnect to create different topologies between routers, and enables the reuse of the metal resources by other FPGA logic when unused by the NoC. We lose this reconfigurability with hard wires but they are, on average, 2.4× faster and consume 1.4× less power than soft wires. We can also trade excess speed for power efficiency by using lower-voltage wires as seen from the "Hard 0.9V" plots.

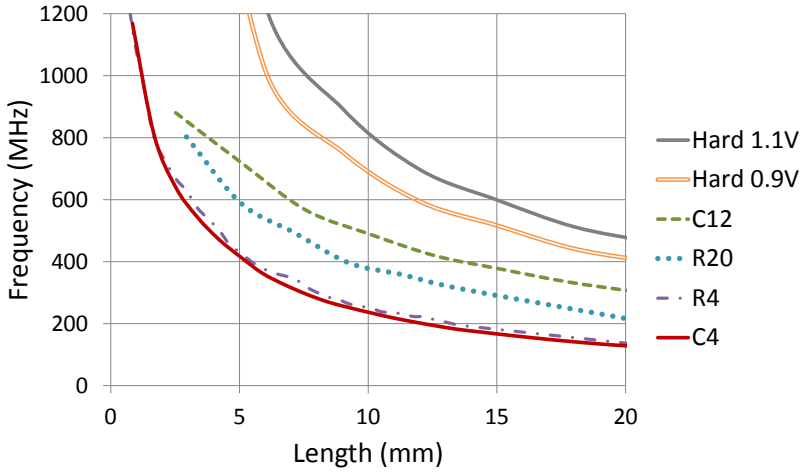


Figure 5.10: Hard and soft interconnect wires frequency.

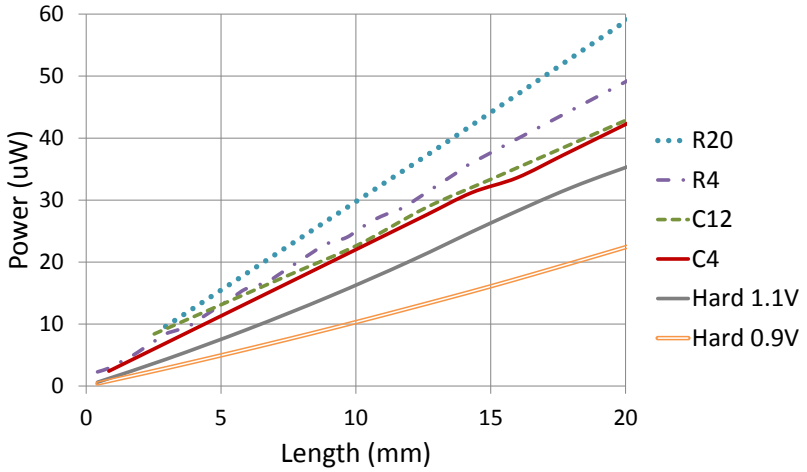


Figure 5.11: Hard and soft interconnect wires power consumption at 50 MHz and 15% toggle rate.

A detailed look at the different soft wires shows that long wires (C12, R20) are faster, per mm, than short wires (C4, R4). Additionally there is a directional bias for power as the horizontal wires (R4, R20) consume more power per mm than vertical ones (C4, C12). An important metric is the distance that we can traverse between routers while maintaining the maximum possible NoC frequency. This determines how far we can space out NoC routers without compromising speed. In the case of soft links and a soft (programmable) clock network, the clock frequency on Stratix III is limited to 730 MHz. At this frequency, short wires can cross 3 mm while longer wires can traverse 6 mm of chip length between routers. When using hard links, we are only limited by the routers' maximum frequency, which is approximately 900 MHz. At this frequency, hard links can traverse 9 mm at 1.1 V or 7 mm at 0.9 V. Although lower-voltage wires are slower, they conserve 40% dynamic power compared to wires running at the nominal FPGA voltage.

Chapter 6

Embedded NoC Options

Contents

In this chapter we combine the component-level results from Chapter 5 to investigate complete NoC systems that are suitable for FPGAs. The first obvious choice is to implement the entire NoC using soft logic – we summarize the design recommendations that we found for soft NoCs. Another option is to harden some of the NoC components and leave others soft – we call this a mixed NoC. Finally, it is also possible to harden an entire NoC, leading to the most efficient but least configurable option.

We discuss the tradeoffs between the three implementation options – soft, mixed and hard – and perform system-level analysis to quantify the overhead in a modern FPGA. We also compare the *raw* efficiency of our NoC options to the simplest form of soft FPGA point-to-point links. This allows us to understand how NoCs fare when they are used for the simplest of interconnection patterns on an FPGA. Also, by comparing to point-to-point links, we understand the *lower bound* of the cost of communication on FPGAs, and it is informative to know how different NoC options compare to that bound.

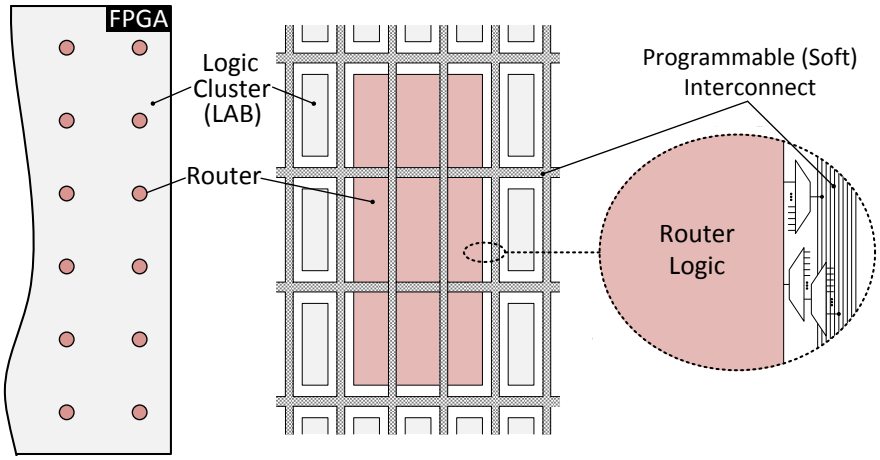


Figure 6.1: Floor plan of a hard router with soft links embedded in the FPGA fabric. Drawn to a realistic scale assuming the router occupies the area equivalent to 9 LABs.

6.1 Soft NoC

Soft NoCs require no architectural changes to the FPGA because they are configured out of the existing FPGA fabric. As such, the strengths of a soft NoC lie in its reconfigurability. We summarize the design recommendations for a soft NoC that makes efficient use of the FPGA's silicon area:

1. BRAM was most efficient for memory buffer implementation even for shallow buffers, so buffer depth is free until the BRAM is full.

2. To increase bandwidth, it is more efficient to increase the flit width rather than the number of ports or VCs.

3. The number of ports and number of VCs scale poorly on FPGAs because of the quadratic increase of allocator and crossbar area.

However, because of their high area overhead and modest operating frequency, soft NoCs are unlikely to replace current interconnect solutions, such as buses or point-to-point links. This is especially true for high bandwidth and streaming applications where both throughput and latency are a concern. We now look at the gains of hardening NoC components. One viable option is to harden the crossbar and allocators and leave the input and output modules soft. This solution moves the critical path from the switch allocator to the input module allowing the router to run at 386 MHz compared to 167 MHz for a fully soft implementation. Such a heterogeneous router occupies 2.34 mm^2 for the baseline parameters. The 1.8× area improvement over a soft implementation is, however, unconvincing. Furthermore we have not yet accounted for the area of the interconnect ports; that is, the switch and connection blocks that would route wires into, out of and around the hard component. For that reason we look more closely into using completely hard routers; the first (and less-invasive) option is to use hard routers with the programmable soft interconnect, and the other option is to build dedicated hard links to connect hard routers.

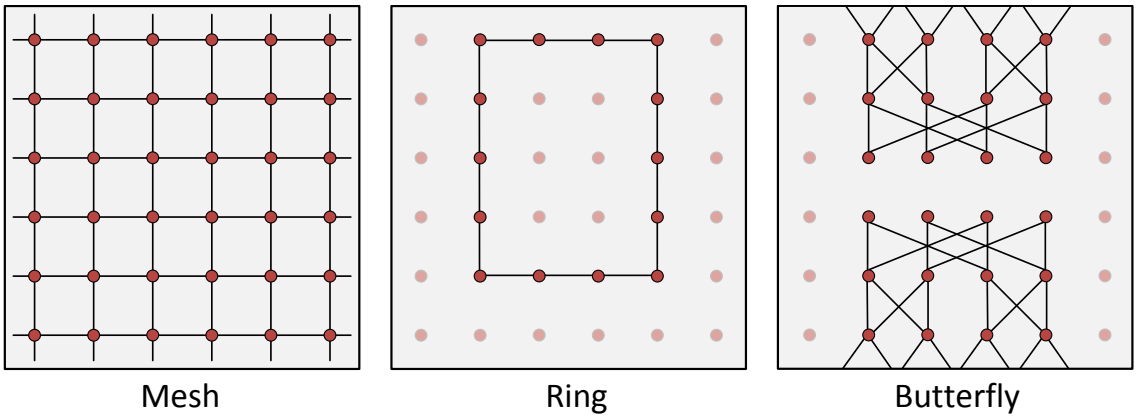


Figure 6.2: Examples of different topologies that can be implemented using the soft links in a mixed NoC.

6.2 Mixed NoCs: Hard Routers and Soft Links

In this NoC architecture, we embed hard routers on the FPGA and connect them via the soft interconnect. While this NoC achieves a major increase in area-efficiency and performance versus a soft NoC, it remains highly configurable by virtue of the soft links. The soft interconnect can connect the routers together in any network topology as shown in Figure 6.2, subject only to the limitation that no router can exceed its (prefabricated) port count. To accommodate different NoC topologies, routing tables inside the router control units are simply reprogrammed by the FPGA CAD tools to match the new topology.

Figure 6.1 shows a detailed illustration of an embedded router connected to the soft interconnect. Note that we must ensure that a sufficient number of interconnect wires intersect the hard router to connect to all of its inputs and outputs. This prevents any interconnection "hot spots" that would over-stress the FPGA's wiring; we aim to have the same interconnection flexibility with NoC routers as we do with LABs. We achieve this by ensuring:

- Connection blocks and switch blocks are only present on the router perimeter. For example the router in Figure 6.1 can only connect to 8 connection/switch blocks because 8 LABs are on its perimeter (although its area equals 9 LABs). This makes physical layout much simpler.
- Hard routers do not over-stress the soft interconnect; they cannot have more inputs/outputs per unit of perimeter than regular LABs.
- Hard routers have equivalent interconnection flexibility as LABs. This implies using 2 levels of programmable multiplexers on each input and one level of programmable multiplexers between an output and a soft interconnect wire.
- Soft wires continue across hard routers but cannot start or end within the router area, only at its perimeter.

6.2.1 Area and Speed

Using the component-level results in Chapter 5, we compute the area and performance of an NoC with hard routers and soft links. Similarly to LABs or BRAMs on the FPGA, a hard router requires

Table 6.1: Soft interconnect utilization for a 64-node 32-bit mixed NoC using either C4/R4 or C12/R20 wires on the largest Stratix III device. Router area is 10 LABs, floorplanned as two rows and five columns.

		Short Wires		Long Wires	
		C4	R4	C12	R20
Demand	–	14,688	19,584	4,896	4,896
Supply	Regional	166,400	159,936	8,320	4,704
	Chip-wide	639,360	1,074,944	34,336	34,944
Utilization	Regional	7.8%	10.9%	58.9%	104%
	Chip-wide	2.0%	1.6%	14.3%	14.0%

programmable multiplexers on each of its inputs and outputs to connect to the soft interconnect in a flexible way. The baseline router has 32 data plus 2 backpressure inputs and outputs per port. We assume that we widen the port (using time-multiplexing) connecting to the FPGA fabric to 128 data plus 2 backpressure inputs and outputs; making the sum of input and output ports 532. Therefore, the total area of the router and the interconnect multiplexers is $0.14\,mm^2 + 532 \times 120\,\mu m^2 = 0.21\,mm^2$, which is equivalent to 9.5 LABs, rounded up to 10 LABs. Assuming that the hard router occupies the area of 5×2 LABs, its perimeter can connect to the equivalent of 10 LABs (or 520 inputs and 400 outputs). This is more than enough to supply the 266 inputs and 266 outputs required by the router, ensuring that the soft interconnect is not stressed; on the contrary, this router has lower interconnect stress than a regular LAB on the FPGA. Note that routers only become input/output pin limited when data width is greater than 220 bits and 4:1 time-division multiplexing (TDM) is used. We repeat these area calculations for all of the design space and take the geometric average; there is a 20× area improvement over soft router implementations (as opposed to 30× when we excluded interconnect).

The speed of an NoC with hard routers and soft links is limited by the soft interconnect and the fabric clock network. We choose the FPGA maximum clock network frequency (730 MHz in Stratix III) as the target NoC frequency, and find that short wires can traverse ~2.5 mm at this speed while longer soft wires can traverse ~5 mm (see Figure 5.10). The FPGA's core dimensions are ~21 mm in each dimension; therefore, an 8×8 mesh of hard routers using the soft interconnect would allow operation at the maximum frequency of 730 MHz even when using the slower short wires.

6.2.2 FPGA Silicon and Metal Budget

To see the cost of a complete system, consider hardening a 64-node NoC on the FPGA. Using baseline parameters, this will occupy area equivalent to $10\,LABs \times 64\,nodes = 640\,LABs$. The total core area of the largest Stratix III FPGA is 412 mm^2 [145]. Therefore, a 64-node hard NoC composed of state-of-the-art VC routers will occupy 3.3% of the core area (~2.2% of total chip area) of this FPGA, compared to 64% of the core area (~43% of total chip area) for a soft implementation.

We now estimate the soft interconnect stress caused by this NoC. Table 6.1 compares the demand for soft wires by NoC links to both the total wire supply and the supply of wires in NoC regions. We define NoC regions as the interconnect channels between routers, and in this example we assume the routers are configured in a mesh topology. In this case NoC links can be constructed most efficiently in the "NoC regions" by concatenating interconnect resources in a straight vertical or horizontal path between routers. We evaluate two cases: making the soft links with short C4/R4 wires, or with the

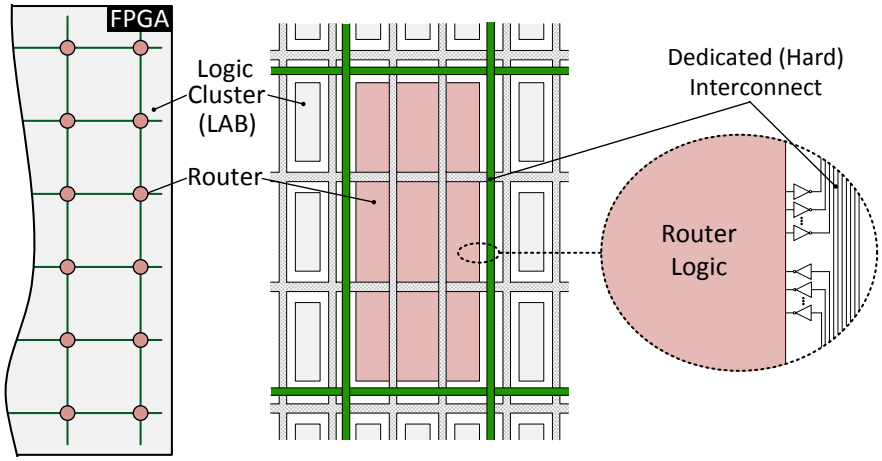


Figure 6.3: Floor plan of a hard router with hard links embedded in the FPGA fabric. Drawn to a realistic scale assuming the router occupies the area equivalent to 9 LABs.

longer C12/R20 wires. Our baseline routers have 32 bidirectional links between channels and 4 bits of flow control for a total of 68 bit-links between two routers. If we only use short wires each bit-link is constructed by stitching together 3 C4 wires in the vertical direction or 4 R4 wires in the horizontal direction. We compute the resulting total interconnect utilization in Table 6.1; note that the NoC links require less than 2% of the total C4/R4 interconnect. The interconnect stress is concentrated in the NoC regions but even there is not excessive; Table 6.1 shows between 8% and 11% of the C4/R4 interconnect is used in these regions. It is difficult to implement NoC links on long wires exclusively; there are not enough R20 wires in NoC regions as shown in Table 6.1.

6.3 Hard NoCs: Hard Routers and Hard Links

Both routers and links are implemented hard for this NoC architecture. Routers are connected to other routers using dedicated hard links; however, routers still interface to the FPGA through programmable multiplexers connected to the soft interconnect. When using hard links, the NoC topology is no longer configurable. However, the hard links save area (as they require no multiplexers) and run at higher speeds compared to soft links, allowing the NoC to run at the routers' maximum frequency. Drivers at the ends of dedicated wires charge and discharge data bits onto the hard links as shown in Figure 6.3.

6.3.1 Area and Speed

We find the complete area overhead of a baseline router including its input and output ports. There are 272 hard links connecting a router to its neighbours and 260 soft interconnect ports connecting the router FabricPort to the FPGA fabric. Therefore, the total area of the router and hard and soft inputs/outputs equals $0.14\,mm^2 + 272 \times 13\,\mu m^2 + 260 \times 120\,\mu m^2 = 0.18\,mm^2$, or 8.3 LABs, rounded up to 9 LABs. We repeat these calculations for all of the design space and take the geometric average; there is a 23× area improvement over soft router implementations (compared to 20× for hard routers and soft links).

The maximum frequency of the baseline router is 943 MHz. At this frequency, hard wires can reach more than one third of the FPGA's dimension (~8mm) as measured in Figure 5.10. Unlike soft links, hard links do not limit the speed of the NoC (to 730 MHz); rather, the routers can operate at their

maximum frequencies and can be spaced-out more if desired. This makes this NoC with hard links 20% faster than the hard NoC with soft links, and 6× faster than a completely soft NoC.

6.3.2 FPGA Silicon and Metal Budget

A 64-node NoC of hard routers and hard links occupies silicon area equivalent to $9\,LABs \times 64\,nodes = 576\,LABs$, or 3.1% of the FPGA core area (~2.1% of total chip area). This is a marginal area improvement over the NoC with hard routers and soft links, but using hard links allows faster NoC operation as well, as we have shown.

However, we should check not only the transistor area but also the metal utilization of hard links. Each hard wire is 2.5 mm long and has a pitch of 1.2 μm. The 64 routers of this 32-bit NoC require a total of 9792 wires; making the total metal area equal $9792 \times 1.2\,\mu m \times 2.5\,mm = 29.4\,mm^2$. The FPGA core area is 412 mm^2, and this is also the area of each metal layer on top of the FPGA core. If 2 metal layers are used for the NoC, then the utilization of each metal layer is only 3.6% for all 9792 wires used in a hard NoC.

6.3.3 Low-Voltage Hard NoC

With hard routers and hard links, an NoC is almost completely disjoint from the FPGA fabric, only connecting through router-to-fabric ports. This makes it easy to use a separate, lower voltage power grid for the NoC, allowing us to trade excess NoC speed for power efficiency. We therefore propose a low-power version of the hard NoC that operates at a lower voltage: for example, 65 nm Stratix III FPGAs use a supply voltage of 1.1 V, while our low-power NoC saves 33% dynamic power by operating at 0.9 V in the same process [4].

6.4 System-Level Power Analysis

This section investigates the power consumed by complete NoCs, especially the mixed and hard options. We investigate how the width of NoC links and spacing of NoC routers affect power consumption, and how that might influence the choice of NoC parameters. Additionally, we report how much of the FPGA's power budget would be spent in these embedded NoCs under worst-case traffic, if they are used for global communication.

6.4.1 Power-Aware NoC Design

Figure 6.4 shows the total dynamic power of mixed and hard NoCs as we vary the width. When we increase the width of our links we also reduce the number of routers in the NoCs to keep the aggregate bandwidth constant at 250 GB/s. For example, a 64-node NoC with 32-bit links has the same total bandwidth as a 32-node NoC with 64-bit links. However, with fewer routers the links become longer so that the whole FPGA area is still reachable through the NoC, albeit with coarser granularity. We assume that our NoCs are implemented on an FPGA chip whose core is 21 mm in each dimension as in the largest Stratix III device [145].

The power-optimal NoC link width varies by NoC type as Figure 6.4 shows. The most power-efficient mixed NoC has 32-bit wide links and 64 nodes. However, for hard NoCs the optimum is at 128-bit width and 16 router nodes. The difference between the two NoC types is a result of the relative routers-to-links

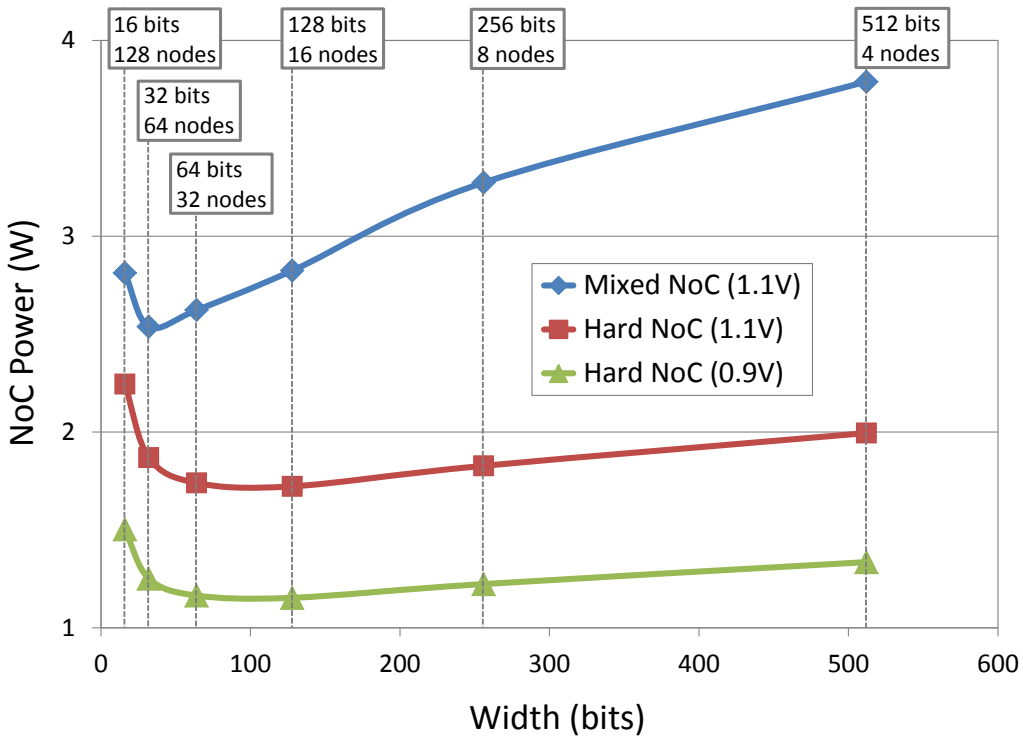


Figure 6.4: Power of mixed and hard NoCs with varying width and number of routers at a constant aggregate bandwidth of 250 GB/s.

power. With fewer but wider nodes, the total router power drops as the control logic power in each router is amortized over more width and hence more data. However, the link power increases since longer wires are used between the more sparsely distributed router nodes. Because soft links consume more power than hard links, they start to dominate total NoC power earlier than hard links as shown in Figure 6.4.

Figure 6.5 shows the NoC power dissipated in routers compared to links for a 64-node NoC. On average, soft links consume 35% of total NoC power, while hard links consume 26%. For NoCs with fewer nodes (and hence longer links), the relative percentage of power in the links is higher.

6.4.2 FPGA Power Budget

We want to find the percentage of an FPGA's power budget that would be used for global data communication on a hard NoC. We model a typical, almost-full[1] FPGA using the Early Power Estimator [14]. The largest Stratix III FPGA core consumes 20.7 W of power in this case, divided into 17.4 W dynamic power and 3.3 W static power. Note that 57% of this power is in the interconnect, while 43% is consumed by logic, memory and DSP blocks.

Aggregate (or total) bandwidth is the sum of available data bandwidth over all NoC links accounting for worst-case contention. A 64-node mixed NoC can move 250 GB/s around the FPGA chip using 2.6 W, or 15% of the typical large FPGA dynamic power budget of 17 W. A hard NoC is more efficient

[1]Only core power is measured excluding any I/Os. We assume that our full FPGA runs at 200 MHz, has a 12.5% toggle rate, and is logic-limited. 90% of the logic is used, and 60% of the BRAMs and DSPs.

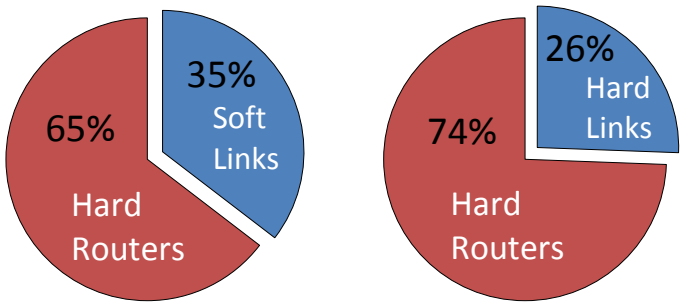


Figure 6.5: Power percentage consumed by routers and links in a 64-node mixed/hard mesh NoC.

and consumes 1.9 W or 11% at 1.1 V and 1.3 W or 7% at 0.9 V. This implies that only 3-6% of the FPGA power budget is needed for each 100 GB/s of NoC communication bandwidth.

To put this in context, 250 GB/s is a large aggregate bandwidth. A single 64-bit DDR3 interface running at the current maximum frequency supported by any FPGA of 933 MHz, produces a maximum data rate of 14.6 GB/s. A PCIe Gen3 x8 interface produces 8.5 GB/s of data in each direction. If this data is transferred to various masters and slaves located throughout the entire FPGA, the average distance traveled is half the width or height of the chip, or 4 routers. Hence an aggregate NoC bandwidth of (14.6×4)+(8.5×2×4)=126 GB/s can distribute the maximum data from these high-speed interfaces throughout the entire FPGA chip.

6.5 Comparing NoCs and FPGA Interconnect

We suggest the use of NoCs to implement global communication on the FPGA; as such, we must compare to existing methods. There are two main types of communication that can be configured on the FPGA as shown in Figure 6.6. The first uses only soft wires to implement a direct point-to-point connection between modules or to broadcast signals to multiple compute modules. The second type of communication uses wires, multiplexers and arbiters to construct logical buses. This is often used to connect multiple masters to a single slave, for example connecting multiple compute modules to external memory. Although the proposed NoCs can implement both of these communication requirements (point-to-point and arbitration), we compare our NoC area with the simplest alternative: point-to-point links that are equal in length to a single NoC link between two routers. Soft point-to-point links consist of FPGA wires that transport data a distance equivalent to a single NoC link. We assume large packets on the NoC, so that the overhead of a packet header is negligible. Nevertheless, this comparison favors the FPGA links, because NoCs can move data anywhere on the chip as well as perform arbitration, while the direct links are limited in length to an NoC link and can perform no arbitration or switching. This simplest form of communication serves as a *lower bound* of any communication overhead.

6.5.1 Area per Bandwidth

As a generalization of the area-delay product, we compute the area overhead of NoCs (or other communication methods) per supported data bandwidth. This figure of merit quantifies the area cost of each Terabyte-per-second of data bandwidth on different communication architectures. The aggregate band-

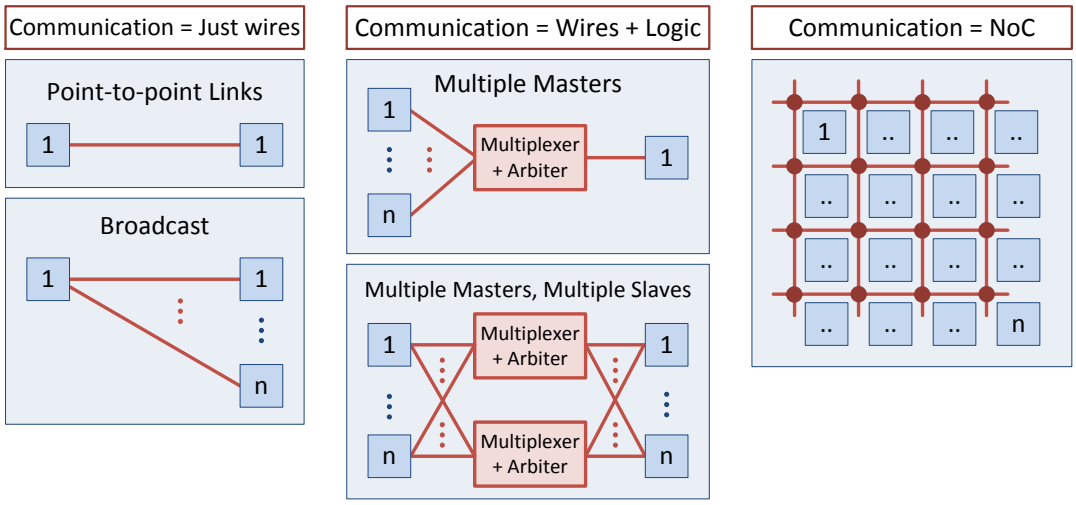


Figure 6.6: Different types of on-chip communication.

width of point-to-point links is simply the product of the link width and its speed. To find the aggregate bandwidth of NoCs we perform a cycle-accurate simulation of NoC routers using ModelSim and attempt to inject packets randomly on each cycle at each port; this represents worst-case uniform-random traffic. Naturally the router stalls due to switch contention and limited buffer space thus limiting bandwidth. We measure this steady-state worst-case bandwidth and report it for different NoC variants in Table 6.3.

We compute a *lower bound* of the area required for communicating data on a conventional FPGA by analyzing the simplest form of communication: point-to-point links using the soft interconnect. To move 250 GB/s of data, one could use 10,000 soft wires running at a reasonable 200 MHz clock frequency. Each unit of data must be transmitted 2.5 mm, the length of one NoC link, which requires stitching four R4 soft wires together. The silicon area overhead of these wires comprises the soft interconnect multiplexers in the switch blocks and logic block inputs. For 10,000 bit-links that consist of 4 R4 wires each, we require at least 200 switch blocks ($\frac{10{,}000\,bits \times 4\,wires\,long}{200\,wires\,switch\,capacity}$), based on our estimate that each switch block has an achievable routing capacity of 200 signals. We then need to connect 10,000 wires to LABs; each of which has 52 inputs. Therefore the input multiplexers of 193 LABs are also taken into account as area overhead. 50% of a LAB's area is interconnect with 30% being associated with input multiplexers and another 20% in the switch blocks and outputs [98]. Using this information we can then estimate the total soft interconnect area as 2.2 mm^2. This translates into 8.8 mm^2 of silicon area to support 1 TB/s of data bandwidth.

A completely soft NoC can be configured onto the FPGA fabric without any architectural changes but a 64-node soft NoC consumes about half the area of an FPGA. Furthermore, it has a low aggregate bandwidth owing to its modest clock frequency as shown in Table 6.3. This leads to the prohibitively high area-per-BW of 4960 mm^2/TBps.

Next, we look at hard NoCs. A hard NoC with soft links is limited to the maximum speed of the FPGA interconnect; nevertheless, this is enough to push this NoC's aggregate bandwidth to 238 GB/s. The total area of this NoC is also greatly reduced compared to soft NoCs (~20×) making its area-per-BW 84× lower than soft NoCs, or 59.4 mm^2/TBps. With hard routers and hard links the NoC can run as fast as the routers at 943 MHz raising its aggregate bandwidth to 307 GB/s. The area-per-BW for this NoC is 1.6× lower than hard NoCs with soft links.

Table 6.2: System-level area-per-bandwidth comparison of different FPGA-based NoCs and regular point-to-point links.

FPGA-based NoCs

NoC Type	Description	Area	Bandwidth	Area per BW
Soft 64-NoC	167 MHz, 32 bits, 2 VCs	269 mm^2	54.4 GB/s	4960 mm^2/TBps
Mixed 64-NoC	730 MHz, 32 bits, 2 VCs	14.1 mm^2	238 GB/s	59.4 mm^2/TBps
Hard 64-NoC	943 MHz, 32 bits, 2 VCs	11.3 mm^2	307 GB/s	36.8 mm^2/TBps
Hard 64-NoC	1035 MHz, 32 bits, 1 VC	7.07 mm^2	236 GB/s	30.0 mm^2/TBps
Hard 64-NoC	957 MHz, 64 bits, 1 VC	10.1 mm^2	437 GB/s	23.1 mm^2/TBps

Conventional Point-to-Point FPGA Interconnect

FPGA Interconnect	Description	Area	Bandwidth	Area per BW
C4 and R4	200 MHz, 10000 bits	2.2 mm^2	250 GB/s	8.8 mm^2/TBps

Some have suggested that VCs consume area and power excessively [78]. We investigate a one-VC version of our hard NoC with hard links and find that it does, in fact, improve area-per-BW. Moving to one VC increases blocking at router ports, reducing aggregate bandwidth by 23%. However, area drops by 60% resulting in a reduced area-per-BW of only 30 mm^2/TBps. Compared to point-to-point links, a hard NoC with hard links and one VC has ~3× area-per-BW. Keep in mind that point-to-point links can perform no arbitration or switching and can only move data between two specified points, while an FPGA-wide NoC can arbitrate between data packets and move them anywhere on the chip.

Finally, by increasing the flit data width of the NoC from 32 to 64 bits, we double its bandwidth while increasing area by only 61%. This increases area efficiency to 23.1 mm^2/TBps, as the router control logic area is amortized over more data bits. This area-per-BW is only 2.6× higher than that of the conventional FPGA wires (8.8 mm^2/TBps).

The results show that soft NoCs consume much area and are impractical for high-throughput applications on FPGAs; however, they may be useful for control-plane and low throughput purposes. Hard NoCs are two orders of magnitude more efficient than soft NoCs. Additionally, lower VCs and higher data widths are favorable in their implementation. When compared against the overhead of point-to-point links, an efficient hard NoC is only 2.6× larger for the same supported bandwidth. This is by no means a head-to-head comparison because, unlike point-to-point links, NoCs are capable of switching data and arbitrating between multiple communicating modules. However, this comparison against the lower bound puts hard NoCs in perspective and strongly suggests that hard NoCs will exceed the efficiency of more complex types of soft interconnect that can also perform arbitration and switching.

6.5.2 Energy per Data

We calculate the energy per unit of data moved by NoCs as an important figure of merit. This is used to compare the energy efficiency of different hard and soft NoCs. We also compare the energy per data of NoCs to conventional point-to-point links on the FPGA. Although point-to-point links merely connect two modules and are incapable of arbitration and switching between many nodes, this comparison shows how the presented NoCs compare to *best-case* conventional interconnect on the FPGA. We show that we can design a hard NoC that uses approximately the same energy as regular (soft) point-to-point links on the FPGA.

Table 6.3: System-level power, bandwidth and energy comparison of different FPGA-based NoCs and regular point-to-point links.

FPGA-based NoCs

NoC Type	Description	Power	Bandwidth	Energy per Data
Soft 64-NoC	1.1V, 167 MHz, 32 bits, 2 VCs	5.14 W	54.4 GB/s	94.5 mJ/GB
Mixed 64-NoC	1.1V, 730 MHz, 32 bits, 2 VCs	2.47 W	238 GB/s	10.4 mJ/GB
Hard 64-NoC	1.1V, 943 MHz[2], 32 bits, 2 VCs	2.67 W	307 GB/s	8.68 mJ/GB
Hard 64-NoC	0.9V, 943 MHz, 32 bits, 2 VCs	1.78 W	307 GB/s	5.78 mJ/GB
Hard 64-NoC	0.9V, 1035 MHz, 32 bits, 1 VC	1.21 W	236 GB/s	5.13 mJ/GB
Hard 64-NoC	0.9V, 957 MHz, 64 bits, 1 VC	1.95 W	437 GB/s	4.47 mJ/GB

Conventional Point-to-Point FPGA Interconnect

FPGA Interconnect	Description	Power	Bandwidth	Energy per Data
C4,12 and R4,20	1.1V, 200 MHz, 10000 bits	1.18 W	250 GB/s	4.73 mJ/GB

In this section we compare our NoC power consumption with the simplest FPGA point-to-point links. The FPGA point-to-point links consist of a mixture of different FPGA wires that are equal in length to a single NoC link; 10,000 wires running at 200 MHz can provide a total bandwidth of 250 GB/s. Table 6.3 shows the result of this comparison. We start by looking at a completely soft NoC that can be configured on the FPGA without architectural changes. Under high traffic, this NoC consumes 5.1 W of power or approximately one third of the FPGA's power budget. However, because its clock frequency is only 167 MHz, it has a relatively low aggregate bandwidth of 54 GB/s. This means that moving 1 GB of data on this soft NoC costs 95 mJ of energy. Conventional point-to-point links only consume 4.7 mJ/GB; soft NoCs seem prohibitively more power-hungry in comparison.

Next, we look at mixed and hard NoCs. A mixed NoC is limited to 730 MHz because of the maximum speed of the FPGA interconnect; nevertheless, this is enough to push this NoC's aggregate bandwidth to 238 GB/s. Note that we calculate bandwidth from simulations and so we account for network contention in these bandwidth numbers. With hard routers and soft links, this NoC consumes 2.5 W or 10 mJ/GB, which is 2.2× that of point-to-point links.

A hard NoC can run as fast as the routers at 943 MHz raising the aggregate bandwidth to 307 GB/s. The energy per data for this NoC is 8.7 mJ/GB; 1.8× more than conventional FPGA links. In Section 6.3.3 we discussed that this completely hard NoC can run at a lower voltage than the FPGA. When looking at the same hard NoC running at 0.9 V instead of 1.1 V, the energy per data drops to 5.8 mJ/GB; 22% higher than conventional FPGA wires.

Next, we look at the overhead of VCs by investigating a one-VC version of our hard NoC running at 0.9 V. Table 6.3 confirms that supporting multiple VCs does reduce energy efficiency. Moving to one VC increases blocking at router ports, reducing aggregate bandwidth by 23% to 236 GB/s. However, power drops by 35% resulting in a reduced energy per data of only 5.1 mJ/GB, a mere 8% higher than the conventional FPGA wires.

Finally, by increasing the flit width of the NoC from 32 to 64 bits, we double its bandwidth while increasing power by only 61%. This increases energy efficiency to 4.5 mJ/GB, as the router control logic

[2] 1.1 V routers can exceed 943 MHz as this conservative frequency measurement is achieved at 0.9 V.

power is amortized over more data bits. This energy per data is *6% lower* than that of the conventional FPGA wires (4.7 mJ/GB).

These findings lead to two important conclusions. First, the most energy-efficient NoC avoids VCs, uses a wide flit width, has hard links and a reduced operating voltage. Second, an embedded hard NoC with hard links on the FPGA can match or even exceed the energy efficiency of the simplest FPGA point-to-point links. This means that a hard NoC, integrated within the FPGA fabric, can implement global communication more efficiently than any soft interconnect that includes arbitration and switching. Hard NoCs are not only area-efficient and fast [2], but energy efficient as well.

6.6 Summary of Mixed and Hard NoCs

Table 6.4 highlights the main characteristics of mixed and hard NoCs. For system-level results, we use a 64-node NoC with baseline (as in Table 4.1) router parameters. Hard NoCs are generally more efficient and can support more communication bandwidth because of their higher speed. Additionally, a very desirable feature in hard NoCs is that the routers are disjoint from the FPGA fabric. This allows us to run the NoC at a lower voltage as discussed in Section 6.3.3. Additionally, it allows us to decouple the NoC's frequency from the FPGA fabric, and thus its timing closure can be completed by the FPGA architect, making it more predictable. Mixed NoCs have the advantage of a configurable topology which allows for greater flexibility in FPGA designs; however, their frequency is tied to the achievable frequency of the FPGA fabric. Overall, both mixed and hard NoC options are quite efficient and only consume ~2% of the area of a modern FPGA. To transport 100 GB/s of data bandwidth, mixed and hard NoCs consume between 2.8–6% of the FPGA power budget as summarized in Table 6.4.

Table 6.4: Summary of mixed and hard FPGA NoC at 65 nm.

		Mixed NoCs	**Hard NoCs**
Description		Hard routers, soft links	Hard routers, hard links
Special feature		Configurable topology	Low-voltage (low-V) mode
Comparison to soft NoCs	Area	20× smaller	23× smaller
	Speed	5× faster	6× faster
	Power	9× less power	11× less-power (15× low-V)
Frequency		730 MHz	943 MHz
Critical Path		Soft interconnect	Switch allocator in router
64-node NoC budget as % of largest Stratix III	Silicon area	2.2%	2.1%
	Metal area	1.6% of short wires	3.6% of 2 metal layers
	Max. Power	6% for 100 GB/s	4.4% for 100 GB/s (2.8% low-V)

Chapter 7

Proposed Hard NoC

Contents

We studied the tradeoffs between hard and soft NoC components, and presented complete NoCs that are entirely hard, soft or a mixture of both. In this chapter we leverage those results to develop a realistic NoC design that targets 28 nm FPGAs. We will use this proposed NoC for our application case studies and any further analyses in the remainder of this dissertation.

Figure 7.1 shows how a hard NoC is implemented on an FPGA device. In this illustration, both the router and the links are implemented in hard logic. To connect to the FPGA, a "FabricPort" (discussed in Chapter 8) is used to adapt width and frequency. Additionally, the embedded NoC connects directly to I/O interfaces such as DDR3 memory interfaces and PCIe transceivers.

7.1 Hard or Mixed?

We advocate the hardening of NoC components on the FPGA because of the large potential efficiency gain compared to soft: 20–23× area, 5–6× speed and 9–15× power improvements. Furthermore, we believe that system-level communication, such as that offered by an NoC, exists in any sizable FPGA application, making an embedded NoC a desirable addition to FPGAs. In addition, an embedded NoC can potentially provide timing closure guarantees to important I/O interfaces such as double data rate version x (DDRx) and PCIe thus making the use of these important I/O interfaces much easier. For these three reasons – efficiency, utility and ease-of-use – we believe that a carefully-designed embedded NoC can greatly improve the design of large systems on FPGAs.

We believe that a hard NoC is a better option than a mixed NoC. Other than the higher efficiency gain, a hard NoC is less coupled to the FPGA fabric. This decoupling allows the NoC's frequency to be completely independent of the FPGA design's placement and routing; this makes NoC timing closure guarantees possible, and also allows guaranteed NoC to high-speed I/O interface timing closure using techniques we explore in Section 8.2. Additionally by hardening the "FabricPort" described in Section 8.1, we can ensure that any modules connected through the NoC are also timing-decoupled. We

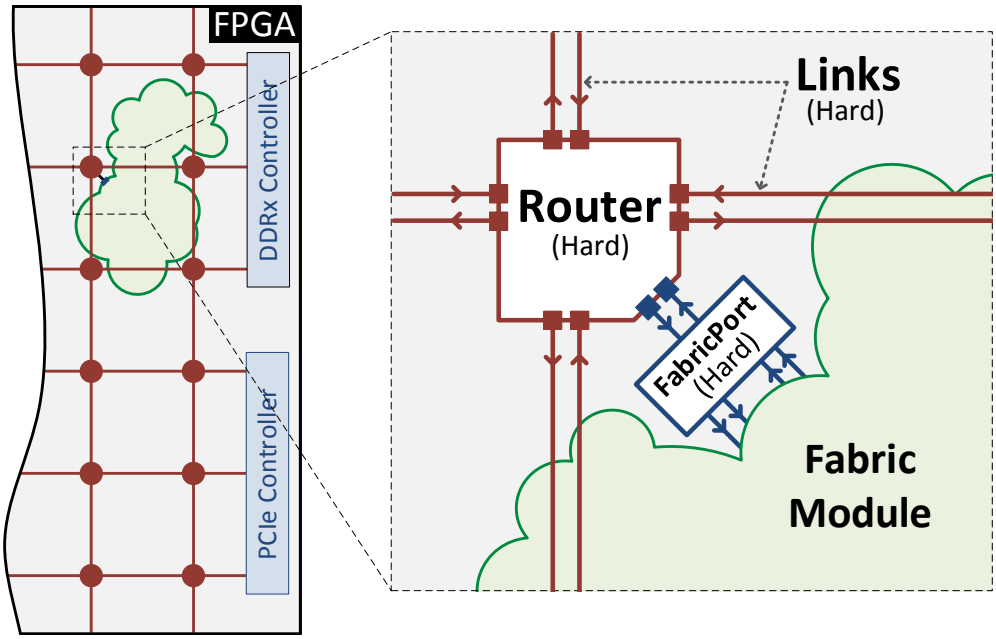


Figure 7.1: Embedded hard NoC connects to the FPGA fabric and hard I/O interfaces.

believe this simplifies the FPGA CAD problem significantly since timing-disjoint application modules can be compiled and optimized independently.

7.2 Design for I/O Bandwidth

A hard NoC must be able to interconnect important I/O and memory interfaces to the FPGA fabric; we look at three of these I/O interfaces on a 28-nm FPGA to motivate the parameters of a viable hard NoC.

DDR3 Interfaces: Port width is typically 64 bits at double data rate (or 128 bits at single data rate), and it can run at 533 MHz, 800 MHz or 1067 MHz. The interface to the FPGA at full bandwidth is ~267 MHz and 512 bits wide.

PCIe Transceivers: A Gen-3 link can have 1, 2, 4 or 8 lanes each running at 8 Gb/s. An 8-lane interface to the FPGA would run at 250 MHz and be 256 bits wide.

Ethernet Ports: 10 Gb/s Ethernet is deserialized on FPGAs into a configurable-width datapath of up to 64 bits at ~150 MHz

Of the three, the interface that requires the highest bandwidth is the DDR3 interface when running at full throughput. To transport DDR3 data, we have 2 options. The first option is to provision the NoC so that we can transport the entire bandwidth of DDR3 memory on one NoC link. This means we must only connect a single router port to the DDR3 memory interface but our NoC links will be quite wide. The second options is to use a fine-grained NoC but we must connect multiple router ports to a single DDR3 memory interface. This more-complicated option might provide higher flexibility, but it complicates high-bandwidth communication considerably – a single DDR3 memory transfer will have to be segmented and reconstructed every time it is transported over a fine-grained NoC. We therefore opt for the first option: a wide coarse-grained NoC that connects to each high-bandwidth interface using

Table 7.1: 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.

a single NoC link. This necessitates that each of the NoC's links must be able to transport the full bandwidth of DDR3 at 16.7 GB/s (1067 MHz×64 bits×2). However, our coarse-grained NoC is not exclusively intended for high-bandwidth wide transfers only – Chapter 8 shows how we build a flexible interface between the NoC and FPGA that allows any data width to be transported over the NoC.

7.3 NoC Design for 28-nm FPGAs

We based most of our investigation in Part I on 65-nm FPGAs. This is mainly because of the availability of the areas of different FPGA blocks, making our analyses more accurate. However, there are newer FPGA devices available at the time of writing this dissertation. From Altera, the Stratix-V 28-nm FPGAs are available. Therefore, our NoC targets these newer devices to make any further investigation more relevant.

To scale area from 65-nm to 28-nm FPGA devices, we assume that our NoC components scale in the same manner as FPGA LABs. That means, if a router uses area equivalent to 10 LABs in a 65-nm Stratix III FPGA, we assume that it will also occupy an area equivalent to 10 LABs in a 28-nm Stratix V FPGA. As for frequency, we assume that a hard router's frequency will scale similarly to other hard blocks on the FPGA. We find that DSP blocks in Xilinx devices increase in frequency by 1.35× from the 65-nm Virtex 5 FPGAs to the 28-nm Virtex 7 FPGAs [146]. We use that 1.35× factor when scaling our NoC frequency from 65 nm to 28 nm devices.

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 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. 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 7.1 summarizes the NoC parameters and properties. We use 2 VCs in our NoC. A second VC reduces congestion by ~30% [4]. We also leverage VCs to avoid deadlock, and merge data streams as we discuss in Chapter 8. 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 – 10 buffer words suffices. With the given parameters, each embedded router occupies an area equivalent to 35 LABs, including the

interface between the router and the FPGA fabric, and including the wire drivers necessary for the hard NoC links [5]. We implement a mesh topology NoC, as it is efficient to layout, and its wiring pattern matches that of the regular interconnect in an island-style FPGA, which we believe will also simplify its integration into the layout flow of an entire FPGA family. As Table 7.1 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).

Part II

Design and Applications

Table of Contents

Chapter 8

FPGA–NoC Interfaces

Contents

We studied the detailed efficiency of hard and soft NoC components in the previous chapters. From the architectural study, we concluded that a hard NoC can be a useful addition to FPGAs because of its efficiency in transporting high bandwidth data across the chip. Additionally, a hard NoC is disjoint from the FPGA fabric allowing us – among other things – to run the NoC at an independent (and very high) clock frequency.

A major question is: how do we connect a hard NoC to the FPGA fabric running almost 4 times slower? More importantly, how can we do that while keeping the interface to the NoC flexible and programmable in a low-cost way? This chapter presents the FabricPort: an incarnation of that FPGA–NoC interface that not only adapts data width and speed between the FPGA logic and a hard NoC, but also does so in a way that is highly configurable. The FabricPort allows us to connect two design modules on the FPGA running at *any* independent width and frequency using an embedded NoC. Essentially, the FabricPort handles the width/frequency conversion, guarantees deadlock freedom and correct ordering of data transfers over an embedded NoC.

Connecting the NoC to I/O interfaces is a different challenge that we investigate in the second part of this chapter. We propose to connect the NoC directly to hard I/O interfaces using IOLinks. While this connection will be different for every I/O interface, we take external memory interfaces as a case study. We suggest an implementation for DDR3/4 memory, and we quantify the exact latency outcome. We also discuss area and power savings at a high level.

8.1 FabricPort

We use the 16-node 150-bit NoC that we proposed in Chapter 7. 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.

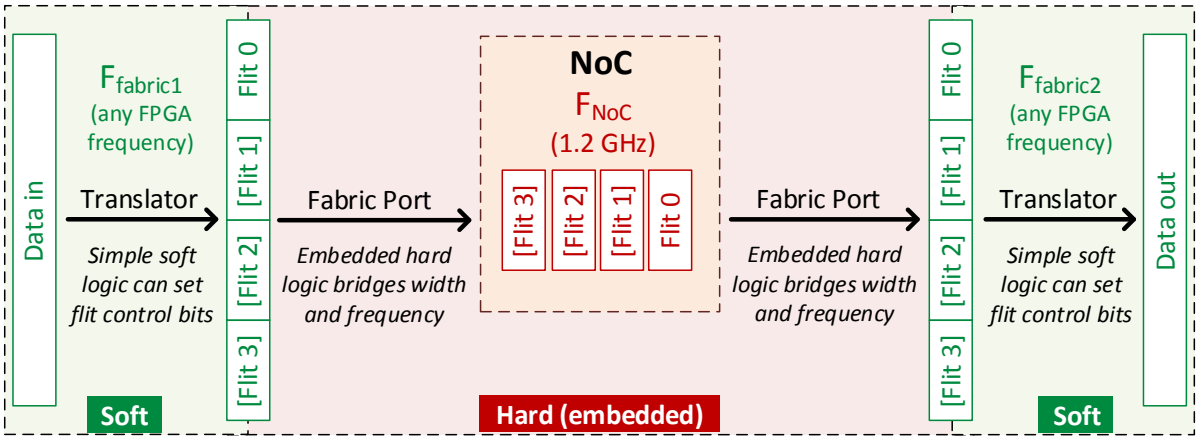


Figure 8.1: Data width and protocol adaptation from an FPGA design to an embedded NoC. Data on the FPGA with any protocol can be translated into NoC flits using application-dependent soft logic (translator). A FabricPort then adapts width (1-4 flit width on fabric side and 1 flit width on NoC) and frequency (any frequency on fabric side and 1.2 GHz on NoC side) to inject/read flits into/from the 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?

Figure 8.1 illustrates the process of conditioning data from any FPGA module to NoC flits, and vice versa. A very simple translator takes incoming data and appends to it the necessary flit control information. For most cases, this translator consists only of wires that pack the data in the correct position and set 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, 5]. 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 is not ready to send/receive data (as Figure 8.2 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.

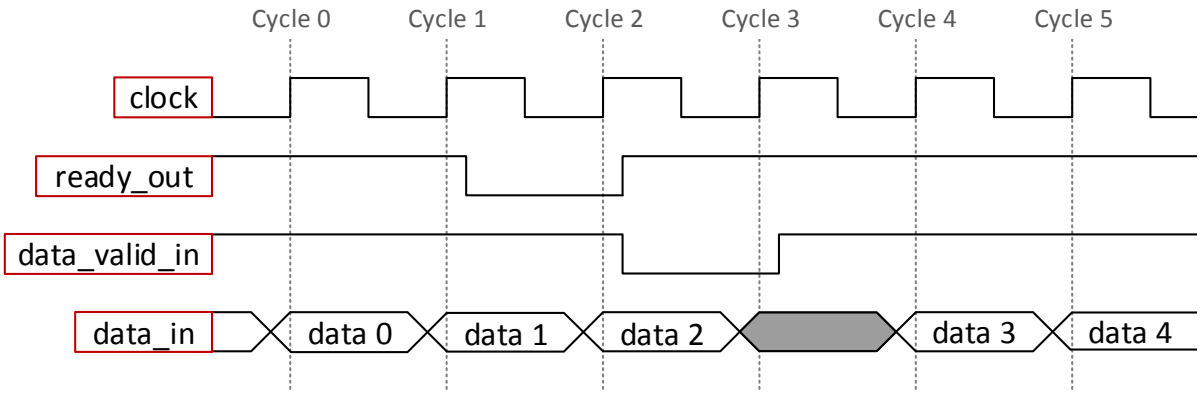


Figure 8.2: Waveform of ready/valid signals between soft module → FabricPort input, or FabricPort output → soft module. After "ready" signal becomes low, the receiver must accept one more cycle of valid data (data 2) after which the sender will have processed the "ready" signal and stopped sending more valid data.

4. **Packet Length**: Send/receive packets of different lengths.

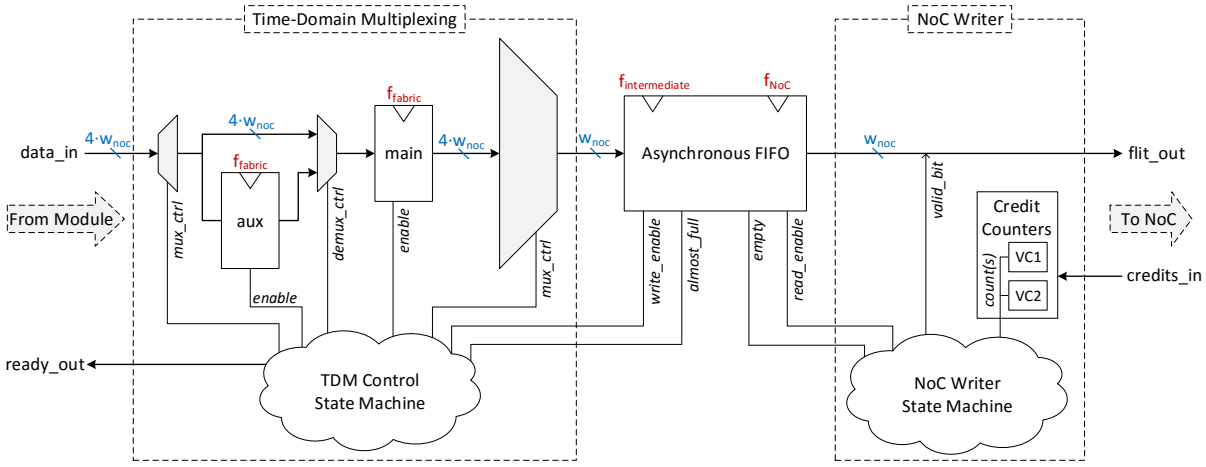
5. **Backpressure Translation**: Convert the NoC's credit-based flow-control system into the more FPGA-familiar ready/valid signals.

8.1.1 FabricPort Input

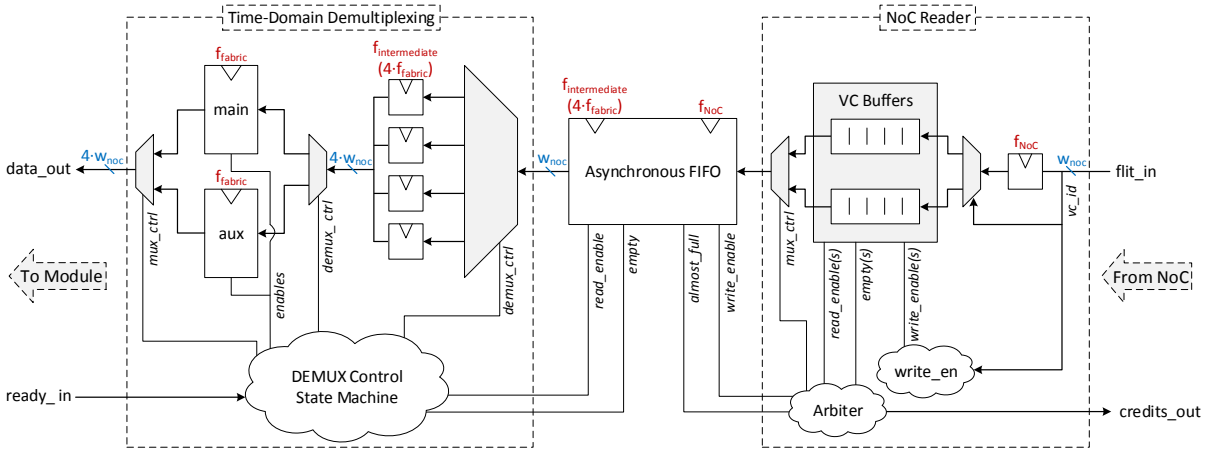
Figure 8.3 shows a schematic of the FabricPort with important control signals annotated. The FabricPort input (Figure 8.3a) connects the output of a module in the FPGA fabric to an embedded NoC input. Following the diagram from left to right: data is input to the time-domain multiplexing (TDM) circuitry on each fabric clock cycle and is buffered in the "main" register. The "aux" register is added to provide elasticity. Whenever the output of the TDM must stall there is a clock cycle before the stall signal is processed by the fabric module. In that cycle, the incoming datum may still be valid, and is therefore buffered in the "aux" registers. To clarify this ready-valid behavior, example waveforms are illustrated in Figure 8.2. 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 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.



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



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

Figure 8.3: FabricPort circuit diagram. 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.

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 [52].

8.1.2 FabricPort Output

Figure 8.3b 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. Figure 8.4 clarifies this point; the upstream router may interleave

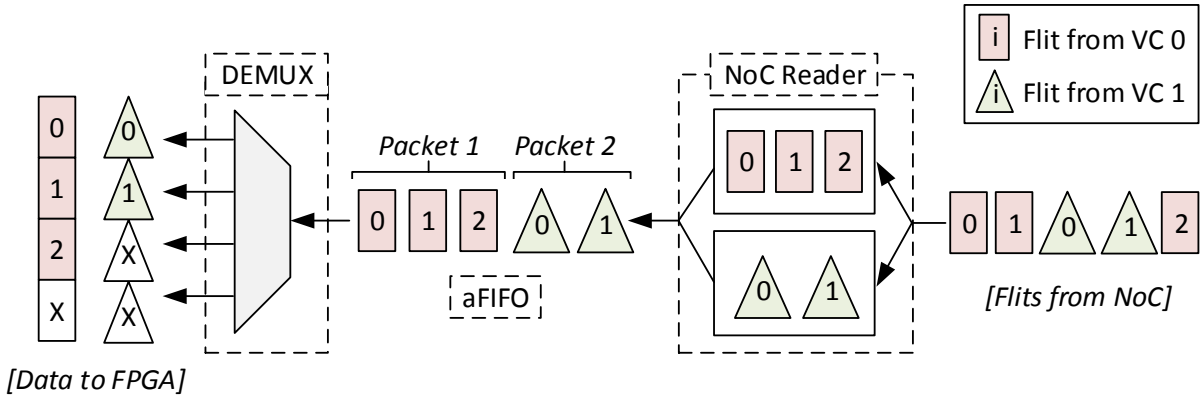


Figure 8.4: FabricPort output sorting flits of different packets. "NoC Reader" sorts flits from each VC into a separate queue thereby ensuring that flits of each packet are contiguous. The DEMUX then packs up to four flits together and writes them to the wide output port but never mixes flits of two packets (except if mixing packets is explicitly enabled in a special *combine-data mode*).

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.

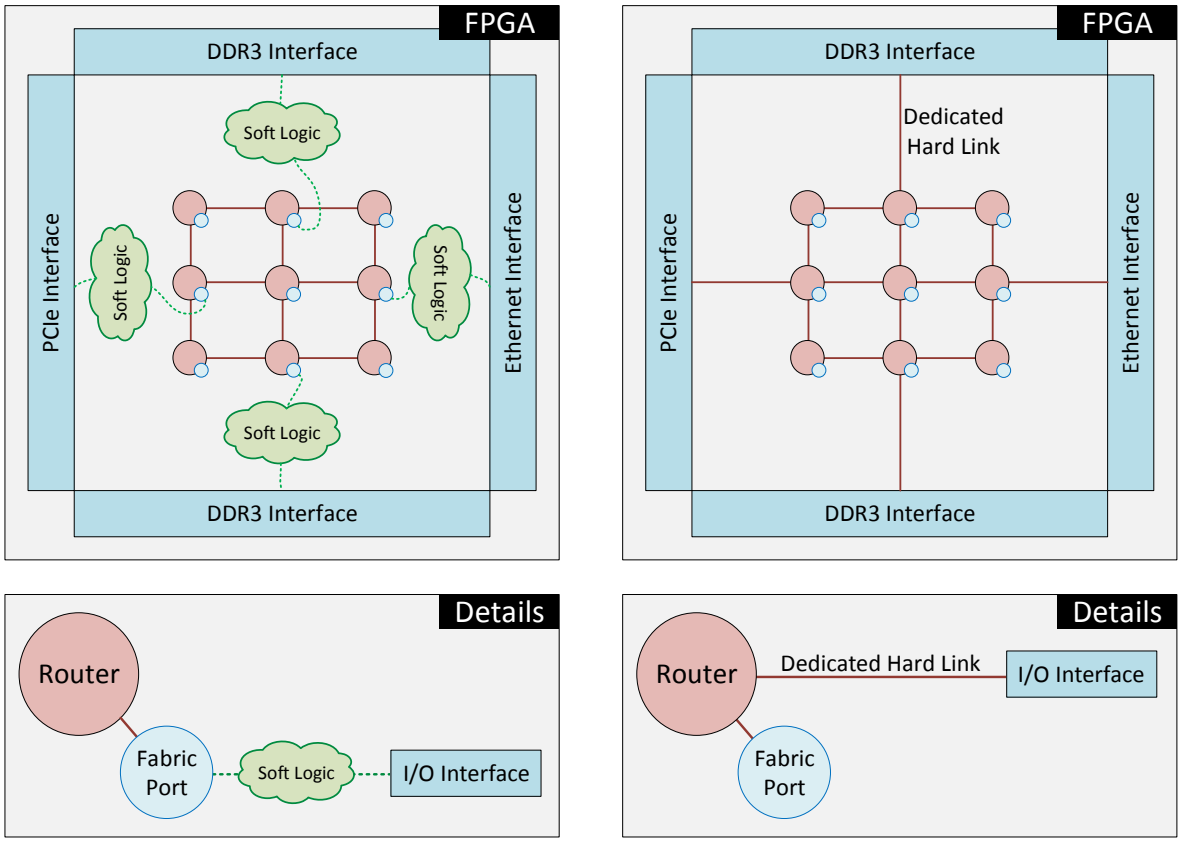
The NoC reader is then responsible for arbitrating between the FIFO queues and forwarding one (entire) packet – one flit at a time – from each VC. We currently implement fair round-robin arbitration and make sure that there are no "dead" arbitration cycles. That means that as soon as the NoC reader sees 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 does not have a number of flits divisible by 4 (see Figure 8.4). This is very much necessary to present a simple interface for designers allowing them to connect design modules to the FabricPort with minimal soft logic.

8.2 IOLinks

The FabricPort interfaces between the NoC and the FPGA in a flexible way. To connect to I/O interfaces, such as external memory interfaces, we can connect through a regular Fabricport interface. This could be done by simply connecting the I/O interface to soft logic which then connects to a FabricPort as shown in Fig. 8.5a. However, the soft logic between an I/O interface and the FabricPort may be difficult to design for many reasons:

- Fast I/O interfaces have very stringent timing requirements, making timing closure on any soft logic connecting to it very challenging.
- The NoC router may be physically placed far away from the I/O interface, thus heavily-pipelined soft logic is required to connect the two. This may incur significant area and power overheads as the data bandwidth of some I/O interfaces is very large, which translates into a wide data path in



(a) Connecting through the FabricPort.

(b) Connecting directly using hard links.

Figure 8.5: Two options for connecting NoC routers to I/O interfaces.

the slow FPGA logic. Furthermore, adding pipeline registers would improve timing but typically worsen latency – a critical parameter of transferring data over some I/Os.

- Any solution is specific to a certain FPGA device and will not be portable to another device architecture.

These shortcomings are the same ones that we use to motivate the use of an embedded NoC instead of a soft bus to distribute I/O data. Therefore, if we connect to I/O interfaces through the FabricPort, we lose some of the advantages of embedding a system-level NoC. Instead, we propose connecting NoC routers directly to I/O interfaces using hard links as shown in Fig. 8.5b. In addition, clock-crossing circuitry (such as an aFIFO) will be required to bridge the frequency of the I/O interface and the embedded NoC since they will very likely differ. We call these direct I/O links with clock crossing "IOLinks". This has many potential advantages:

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

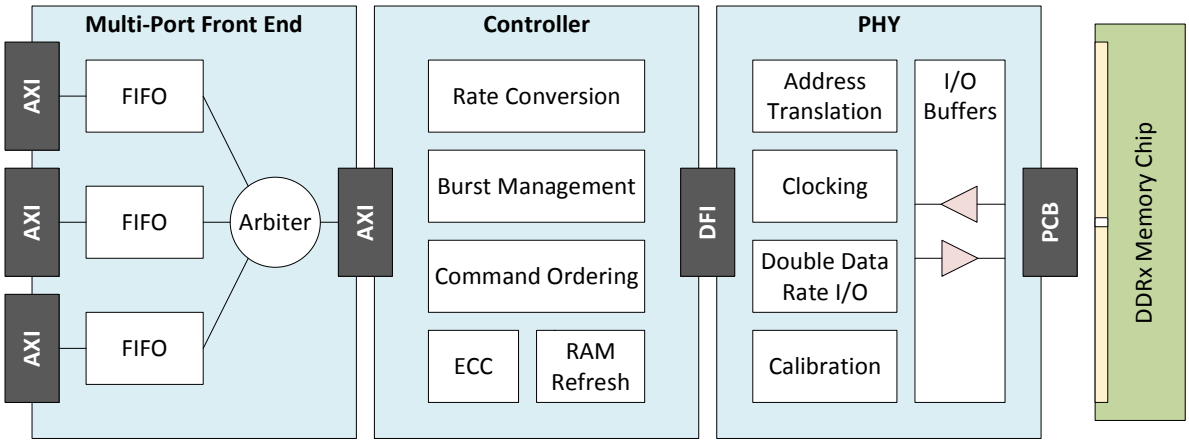


Figure 8.6: Block diagram of a typical memory interface in a modern FPGA.

One possible shortcoming of IOLinks is the loss of configurability. It is important to design these I/O links such that they do not rid the FPGA I/O interfaces of any of their built-in flexibility. We therefore advocate that any use of IOLinks should be optional. Multiplexers in the I/O interfaces can choose between our IOLinks and the traditional interface to the FPGA logic. This will maintain the option of directly using I/O interfaces without using IOLinks, thus maintaining the configurability of I/O interfaces.

To be able to send packets to the directly-connected I/O interfaces, we need to extend NoC addressing to include the directly-connected I/Os. For example, our proposed NoC in Chapter 7 has 16 routers and therefore requires 4 addressing bits in its packet format. However, this NoC can connect up to 16 I/O interfaces along its perimeter (by extending the mesh topology at each side). We therefore have a maximum of 32 different addresses (16 routers and 16 direct I/Os) for this NoC and we will require 5 address bits instead of 4.

8.2.1 DDR3/4 Memory IOLink Case Study

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

Memory Interface Components

Figure 8.6 shows a typical FPGA external memory interface. The physical interface (PHY) is used mainly for clocking, data alignment and calibration of the clock and data delays for reliable operation, and to translate double-rate data from memory to single-rate data on the FPGA. In modern FPGAs, especially the ones that support fast memory transfers, the PHY is typically embedded in hard logic. The PHY presents a *standard*[1] protocol called DFI to the next part of the external memory interface; the memory controller.

[1]Neither Altera nor Xilinx support ddr-phy interface (DFI) fully, but they have a very similar protocol to bridge the PHY and controller [18, 148].

Table 8.1: Typical DDR3/4 memory speeds and their quarter-rate conversion on FPGA.

Memory Frequency	Memory Width	Rate Conversion	FPGA Frequency	FPGA Width
667 MHz	64 bits	quarter	167 MHz	512 bits
800 MHz	64 bits	quarter	200 MHz	512 bits
933 MHz	64 bits	quarter	233 MHz	512 bits
1067 MHz	64 bits	quarter	267 MHz	512 bits
1200 MHz	64 bits	quarter	300 MHz	512 bits
1333* MHz	64 bits	quarter	333 MHz	512 bits

*Supported for DDR4 memory in Xilinx Ultrascale+ at the highest speed grade and voltage only [149].

The memory controller (see Figure 8.6) is in charge of higher-level memory interfacing. This includes regularly refreshing external memory and computing error correcting codes (ECC) if that option is enabled. Additionally, addresses are translated into bank, row and column components, which allows the controller to issue the correct memory access command based on the previously accessed memory word. Importantly, memory controllers also optimize the order of commands to external memory, to minimize the number of costly accesses. An example optimization is the coalescing of memory commands that access the same memory bank or row [148]. This example optimizes latency because it avoids incurring the additional latency of switching between banks and rows. The memory controller is sometimes implemented hard and sometimes left soft, but the trend in new devices is to harden the memory controller to provide an out-of-the-box working memory solution [18]. Some designers may want to implement their own high-performance memory controllers to exploit patterns in their memory accesses for instance, therefore, FPGA vendors always allow direct connection to the PHY, bypassing the hard memory controller. However, hard memory controllers are more efficient and much easier to use making it a more compelling option for most users, especially as FPGAs start being used by software developers (in the context of HLS and data center computing) who do not have the expert hardware knowledge to design a custom memory controller. Therefore, it is our opinion that hard memory controllers will be more commonly used than their soft/custom counterparts.

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

Rate Conversion

One of the functions of an FPGA memory controller is rate conversion. This basically down-converts data frequency from the high memory frequency (~1 GHz) to a lower FPGA-compatible frequency (~200 MHz). All modern memory controllers in FPGAs operate at quarter rate; meaning, the memory frequency is down-converted 4× and memory width is parallelized eightfold[2]. Table 8.1 lists the typically-supported DDR3/4 memory frequencies and their corresponding quarter-rate conversion on FPGAs.

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

Table 8.2: Altera's DDR3 memory latency breakdown at quarter, half and full-rate memory controllers. Latency is shown in full-rate memory clock cycles (adapted from [18]).

	Quarter Rate	Half Rate	Full Rate
Controller Address and Command	20	10	5
PHY Address and Command	8-11	3-4	0
Memory Read	5-11	5-11	5-11
PHY Read Return	14-17	6-7	4
Controller Read Return	8	4	10
Round Trip	57-67	28-36	24-30
Round Trip without Memory	52-56	23-25	19

This quarter-rate conversion is arguably *necessary* to be able to use fast DDRx memory on an FPGA – how else can we transport and process fast memory data at the modest FPGA speed? However, there are both performance and efficiency disadvantages that arise due to quarter-rate conversion in the memory controller.

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

Latency Overhead: Operating at the lower frequency increases memory transfer latency. This is mainly because each quarter-rate clock cycle is much slower (4× slower) than a full-rate equivalent. Table 8.2 shows the breakdown of memory read roundtrip latency for Altera's DDR3 memory controller [18]. It shows that we can improve round-trip latency more than twofold if we use a full-rate memory controller instead of the currently-used quarter-rate memory controllers.

Proposed IOLink

As outlined in the beginning of this chapter, we propose directly connecting an NoC link to I/O interfaces. For the external memory interface, we propose connecting a direct NoC link to the AXI port after the hard memory controller (see Figure 8.6). Note that we also propose implementing a memory controller that supports full-rate memory operations, even at the highest memory speeds. This topology leverages the high speed and efficiency of a full-rate controller, and avoids the costly construction of a MPFE and soft bus to transport the data. Instead, an efficient embedded NoC fulfills the function of both the MPFE and soft bus in buffering and transporting DDRx commands and data. Furthermore it does so at full-rate memory speed with much lower latency.

Table 8.3 details the latency breakdown of a memory read transaction when fulfilled by a current typical memory interface, and an estimate of latency when an embedded NoC is connected directly to a full-rate memory controller. We use the latency of the memory chip, PHY and controller from Table 8.2. For the MPFE, we estimate that it will take at least 2 system clock cycles[3] (equivalent to 8 memory clock cycles) to buffer data in a burst adapter and read it back out – even though this is a very rough estimate, it is also a conservative estimate on the latency of a hard MPFE which performs both buffering and arbitration. As for the soft bus, we generate buses in Altera's Qsys system integration tool with different levels of pipelining. Only highly pipelined buses (3-5 stages of pipelining) can achieve timing

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

Table 8.3: Read transaction latency comparison between a typical FPGA quarter-rate memory controller, and a full-rate memory controller connected directly to an embedded NoC link. Note that latency is measured in full-rate memory clock cycles.

Current System	
Component	Latency
Memory	5-11
PHY (quarter-rate)	22-28
Controller (quarter-rate)	28
MPFE	>8
Soft Bus	24-44
Total	87-119

NoC-Enhanced System	
Component	Latency
Memory	5-11
PHY (full-rate)	4
Controller (full-rate)	15
MPFE	–
Hard NoC	32-68
Total	56-98

closure for a sample 800 MHz memory speed [6]. The round-trip latency of these buses in the absence of any traffic is 6-11 system clock cycles (depending on the level of pipelining).

To estimate the embedded NoC latency in Table 8.3, we used the zero-load latency from Figure 9.1. The round-trip latency consists of the input FabricPort latency, the output FabricPort latency and twice the link traversal latency. At a 300 MHz fabric (system) frequency, FabricPort input latency is ~2 cycles, FabricPort output latency is 3 cycles and link traversal latency ranges between 1.5-6 cycles depending on the number of routers traversed. This adds up to a round-trip latency between 8-17 system clock cycles.

As Table 8.3 shows, use of the embedded NoC can improve latency by approximately 1.5×. Even though the embedded NoC has a higher round-trip latency compared to soft buses, latency improves because we use a full-rate memory controller, and avoid a MPFE. We directly transport the fast memory data using an NoC link, and only down-convert the data at a FabricPort output at the destination router where the memory data will be consumed. This undoubtedly reduces area utilization as well. Even more importantly, this avoids time-consuming timing closure iterations that are necessary whenever we connect to an external memory interface. We quantify the design effort and efficiency (area and power) advantages of using an embedded NoC over a soft bus in Section 11.1.

Chapter 9

Design Styles and Rules

Contents

Having presented the interfaces between the NoC and FPGA, we can start to implement FPGA communication on embedded NoCs. We start this chapter by evaluating the latency and throughput of our proposed NoC including the FabricPort. Next, we discuss important connectivity and design rules that are required for correct NoC communication. Finally we investigate FPGA design styles – both latency sensitive and insensitive design – in the context of embedded NoCs. We show how both design styles can be used with our NoC when the proper design rules are followed.

9.1 Latency and Throughput

Figure 9.1 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 3 NoC clock cycles, 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 Figure 9.1 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

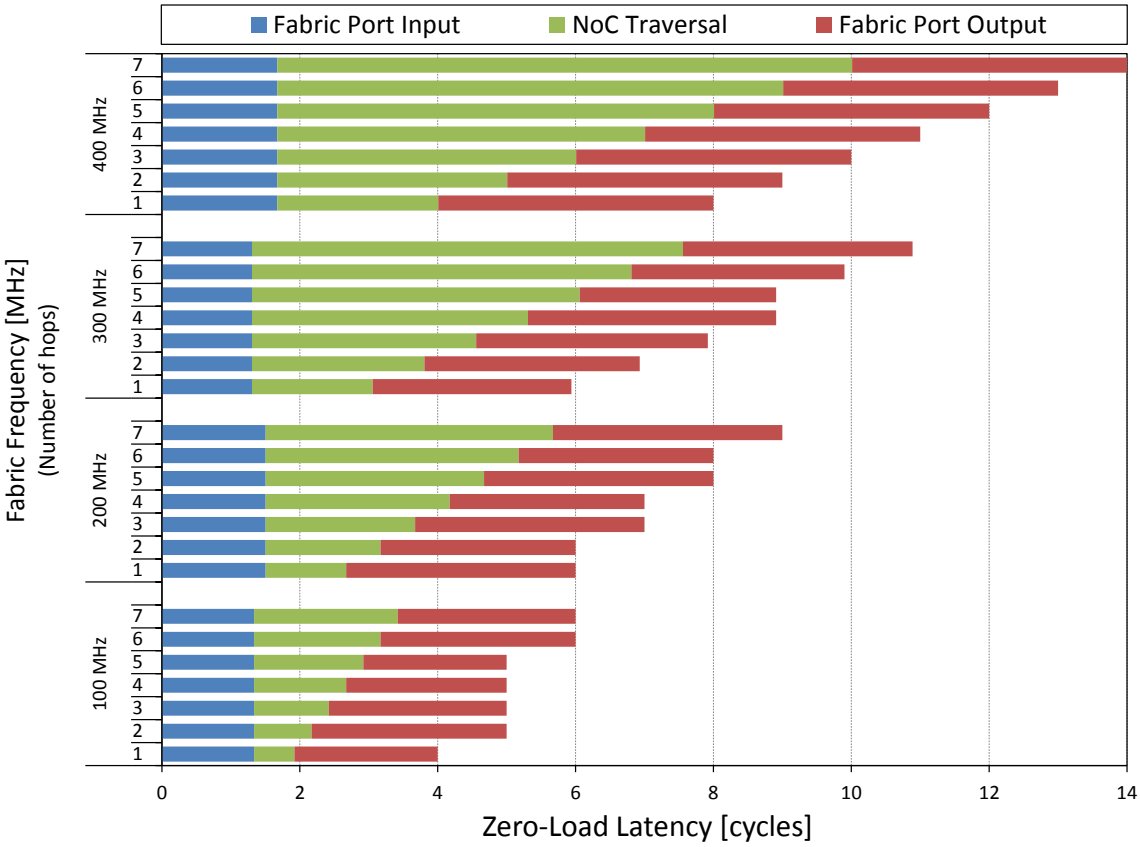


Figure 9.1: Zero-load latency of the embedded NoC (including FabricPorts) at different fabric frequencies. Latency is reported as the number of cycles at each frequency. The number of hops varies from 1 hop (minimum) to 7 hops (maximum – chip diagonal).

output data in the DEMUX unit. Additionally, the latency of the FabricPort is lowest at 300 MHz. This is because 300 MHz is exactly one quarter of the NoC frequency, meaning that the intermediate clock is the same as the NoC clock and the aFIFO reads and writes flits at the same frequency, thus no additional clock-crossing latency is incurred.

Figure 9.2 plots the throughput between any source and destination on our NoC in the absence of contention. The NoC is running at 1.2 GHz with 1-flit width; therefore, if we send 1 flit each cycle at a frequency lower than 1.2 GHz, our throughput is always perfect – we'll receive data at the same input rate (one flit per cycle) on the other end of the NoC path. The same is true for 2-flits (300 bits) at 600 MHz, 3 flits (450 bits) at 400 MHz or 4 flits (600 bits) at 300 MHz. As Figure 9.2 shows, the NoC can support the mentioned width–frequency combinations as each is a different way to utilize the NoC bandwidth.

In 28-nm FPGAs, we believe that very few wide datapath designs will run faster than 300 MHz; therefore the NoC is very usable at all its different width options. When the width–frequency product exceeds the NoC bandwidth, packet transfers are still correct; however, the throughput degrades and the NoC backpressure stalls the data sender. This results in the throughput reduction shown in Figure 9.2.

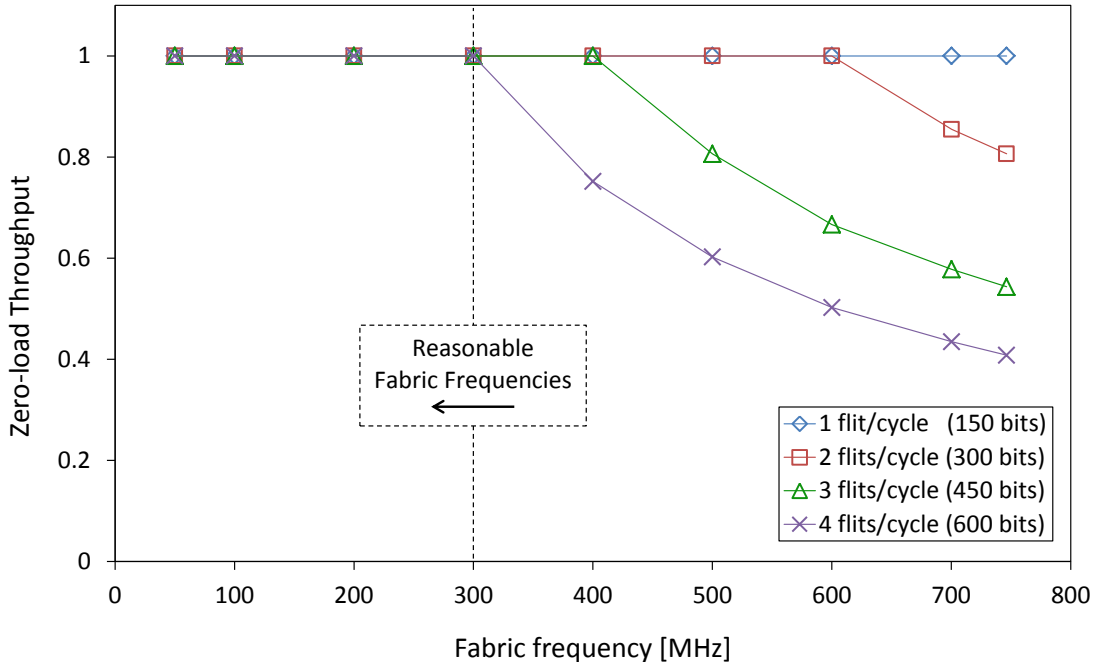


Figure 9.2: Zero-load throughput of embedded NoC path between any two nodes, normalized to sent data. A throughput of "1" is the maximum; it means that we receive i flits per cycle, where i is the number of flits we insert in the FabricPort each cycle.

9.2 Connectivity and Design Rules

This section describes different aspects of the connection of and communication between modules on the NoC. We first describe the packet format before discussing module connectivity using the FabricPort. Next, we look at important design constraints for FPGA communication. We discuss the importance of correct data ordering, and how it is enforced using our embedded NoC. We then we show how to ensure deadlock freedom when using the NoC and FabricPort. Note that the design conditions and rules[1] presented here are necessary, but not sufficient for the intended operation of the NoC.

9.2.1 Packet Format

Figure 9.3 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) [52]. 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.

[1]The design rules we present are meant more as guidelines on how an embedded NoC can be made to suit FPGA communication – there could be other methods covered in the NoC literature [52] to achieve the same intended operation.

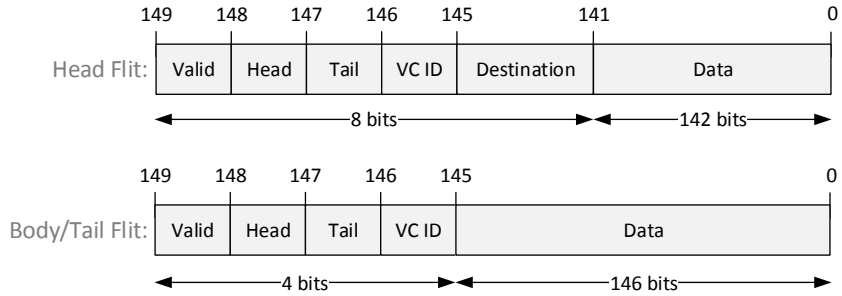


Figure 9.3: NoC packet format. Packets consist of a head flit and zero-or-more body flits. The figure shows flits for a 16-node 150-bit-width NoC with 2 VCs. Each flit has control data to indicate whether this flit is valid, and if it is the head or tail flit (or both for a 1-flit packet). Additionally each flit must have the VC number to which it is assigned and a head flit must contain the destination address.

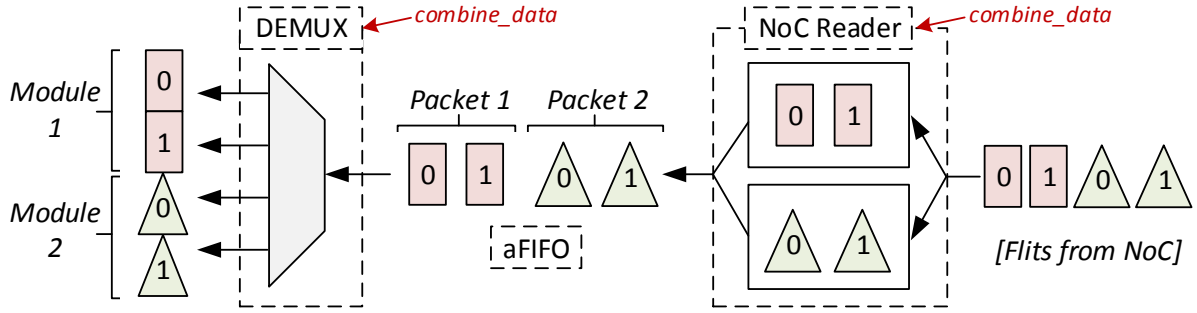


Figure 9.4: FabricPort output merging two packets from separate VCs in *combine-data* mode, to be able to output data for two modules in the same clock cycle.

9.2.2 Module Connectivity

The FabricPort converts 22.5 GB/s of NoC link data bandwidth (150 bits, 1.2 GHz) to 600 bits and any fabric frequency on the fabric side. An FPGA designer can then use any fraction of that port width to send data across the NoC. However, the smallest NoC unit is the flit; so we can either send 1, 2, 3 or 4 flits each cycle. If the designer connects data that fits in one flit (150 bits or less), all the data transported by the NoC is useful data. However, if the designer wants 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 Figure 8.1) 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 Figure 8.4 shows. To overcome this limitation, we create a *combine-data* mode as shown in Figure 9.4. 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.*

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

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

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.

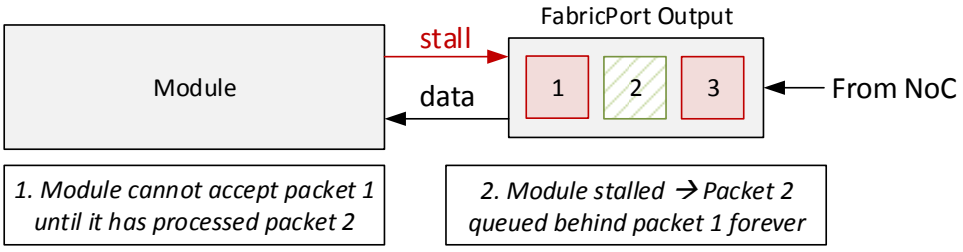
To solve the first problem, we only use routing algorithms, in which routes are the same for all packets that belong to a connection.

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

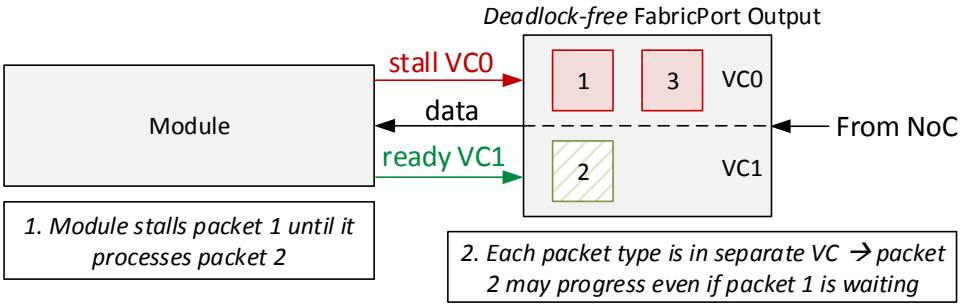
Deterministic routing algorithms such as dimension-ordered routing [52] fulfill Condition 2 as they always select the same route for packets on the same connection.

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 (Figure 9.4), 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.*



(a) Standard FabricPort output.



(b) Deadlock-free FabricPort output.

Figure 9.5: 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.

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 [52]. 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.

9.2.4 Dependencies and Deadlock

Two *message types* may not share a standard FabricPort output (Figure 8.3b) if a dependency exists between the two message types. An example of dependent message types can be seen in video processing IP cores: both control messages (that configure the IP to the correct resolution for example) and data messages (pixels of a video stream) are received on the same port [17]. An IP core may not be able to process the data messages correctly until it receives a control message.

Consider the deadlock scenario in Figure 9.5a. 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 [137]. 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. Figure 9.5b shows that even if there is a dependency between different messages, they can share a FabricPort output provided each uses a different VC. Note that the

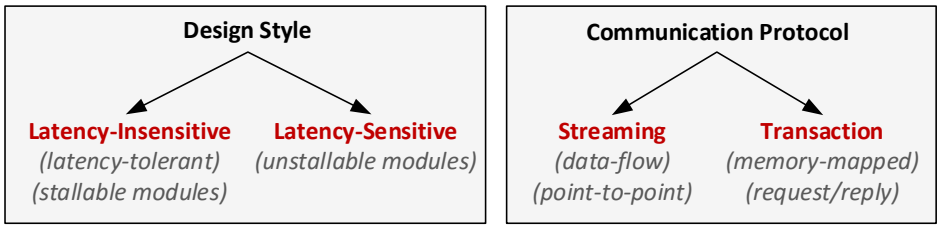


Figure 9.6: Design styles and communication protocols.

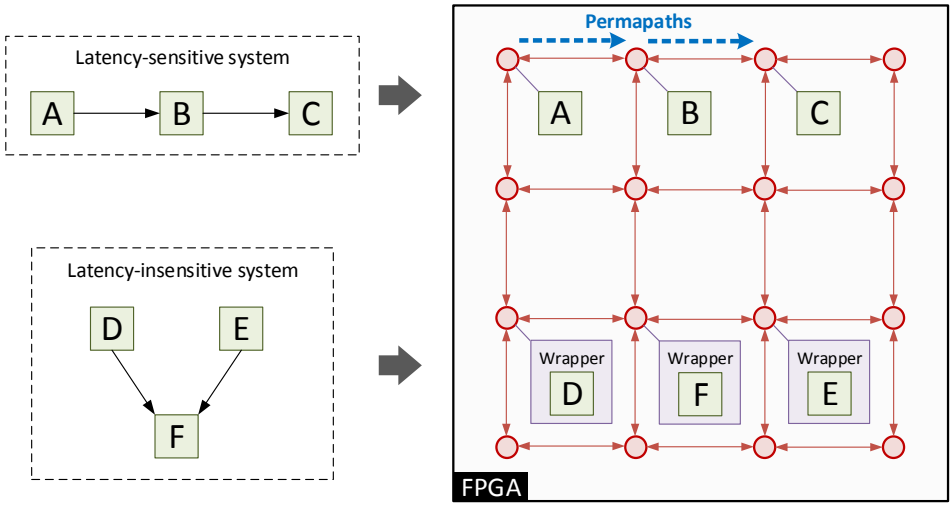


Figure 9.7: Mapping latency-sensitive and latency-insensitive systems onto an embedded NoC. We reserve *Permapaths* on the NoC to guarantee a fixed latency and perfect throughput for a latency-sensitive application. For latency-insensitive systems, modules must be encapsulated with wrappers to add stall functionality.

use of VCs to break protocol-level deadlock will lead to the use of more than 2 VCs in embedded NoC architectures to have sufficient flexibility.

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.*

9.3 Design Styles

Figure 9.6 shows the two possibilities of synchronous design styles, as well as two communication protocols that are common in FPGA designs. The two design styles are "latency-insensitive", and "latency-sensitive". 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 [36]. 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. Either design style can be used with each of the two communication protocols discussed below.

FPGA communication can be broadly classified into either streaming transfers or transactions. Simply put, streaming transfers are one-way messages that are transmitted from a source to a sink. Trans-

actions, however, consists of requests and replies. A master module sends a request message to a slave, which in turn issues a reply message back to the master – a transaction is only complete once the reply reaches the master. Transaction communication is considered a higher-level communication protocol than the simpler streaming communication. This is why typical communication solutions implement transactions as a composition of streaming transfers [85]. We therefore focus on streaming communication in this chapter, and we discuss transaction communication in Chapter 13 in the context of our NoC CAD system. Figure 9.7 illustrates this a sample mapping of latency sensitive and insensitive systems on an FPGA, and highlights that sometimes soft logic wrappers may be necessary. The next sections discuss latency-sensitive and latency-insensitive design with our NoC, and we enumerate rules and conditions that are required to implement each style of communication.

9.3.1 Latency-Insensitive Design with a 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 [36]. 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 [36].

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 Figure 9.7. 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 [115]. 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 [115], and the NoC-compatible wrappers in Figure 9.8 for wrappers that support one input and one output and a width between 100 bits and 600 bits. As Figure 9.8 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. In Chapter 12 we describe such a CAD flow.

9.3.2 Latency-Sensitive Design with a 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 [68]. We analyze the latency

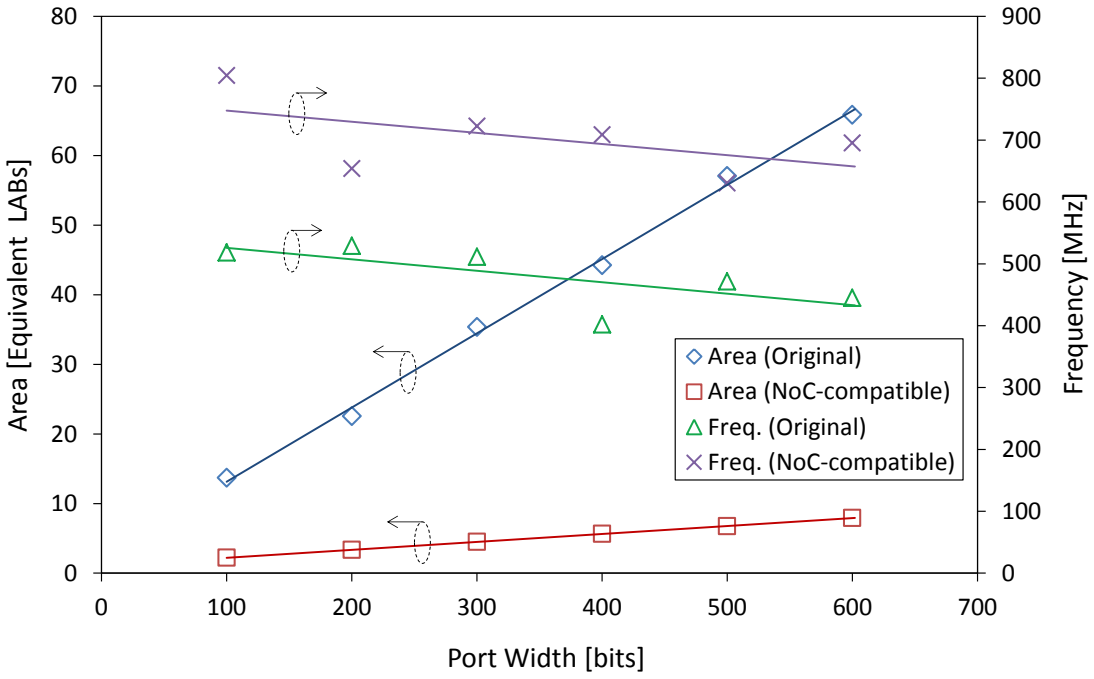


Figure 9.8: Area and frequency of latency-insensitive wrappers from [115] (original), and optimized wrappers that take advantage of NoC buffering (NoC-compatible).

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.

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* [52] – for our system, a buffer depth of 10 flits suffices.

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 for high-throughput applications 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 Figure 9.2 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 in Figure 9.2 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 is 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.

There is one final constraint that is necessary to ensure error-free latency-sensitive transfers. It pertains to "clock drift" that occurs between the intermediate clock and the NoC clock – these are respectively the read and write clocks of the aFIFO in the FaricPort (Fig. 8.3). If these clocks are different, and they drift apart because of their independence, data may not be latched correctly onto the synchronizing registers in the aFIFO resulting in a missed clock edge. While this doesn't affect overall system throughput by any measurable amount, it may corrupt a latency-sensitive system where the exact number of cycles between data transfers is part of system correctness – Condition 7 circumvents this problem.

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

Chapter 10

Prototyping and Simulation Tools

Contents

Before delving into application case studies, we present the tools and methodologies that we developed to properly emulate an embedded NoC. Our architecture prototyping tool `NoC Designer` can easily find the area, delay or power of an embedded NoC, and automatically creates a floorplan of the NoC on a sample FPGA device. To simulate the cycle-accurate behaviour of an application connected to our NoC, we developed `RTL2Booksim`. This simulator connects Modelsim (a hardware simulator) and Booksim (an NoC simulator) thus allowing a complete system-level cycle-accurate simulation. Additionally, we emulate the existence of NoCs on FPGAs to model the physical placement and routing consequences of connecting a design to an NoC in an FPGA device.

10.1 NoC Designer

`NoC Designer`[1] is an online tool to prototype hard, soft or mixed NoCs on FPGAs. It provides a visual front-end for some of the data that we have gathered in our research; area, frequency and power measurements from synthesized NoC implementations. We aim to provide our data for fellow researchers to either use in their own work, or to see the backing data for our research publications. After you specify any NoC configuration, the tool consults its database of measured area, speed, power and bandwidth. If the configuration is not present in our measurements database, we interpolate between our points to compute a reasonable estimate.

We use a weighted linear interpolation to find our estimate. Simply put, our interpolation algorithm searches our results database for all *close* points, and interpolates from each one of those points then multiplies by a weight depending on the distance from that point. We found that this gives a reasonable estimate that is suitable for prototyping purposes. Note that we clearly mark interpolated results in `NoC Designer` to differentiate them from the exact measured results. `NoC Designer` is accessible at: `http://www.eecg.utoronto.ca/~mohamed/noc_designer`.

[1] `NoC Designer` was developed by summer student Ange Yaghi under my supervision.

Figure 10.1: Screenshot of `NoC Designer` showing its three data analysis features.

Figure 10.1 shows a screenshot of `NoC Designer` and highlights its three data analysis views. The first is a system-level table in which a user can enter the NoC parameters: number of nodes, number of ports per router, channel width, number of VCs and buffer depth. The user can also select the implementation option: hard, soft or mixed. The table then computes system-level metrics such as total area and power, frequency of operation, aggregate bandwidth and other important metrics. This allows a quick comparison between NoCs, and makes it easy to spot the major system-level differences. The second feature in `NoC Designer` is a NoC visualization panel. In this panel, a user can visualize the floorplan of any of the NoCs specified in the system-level table. The visualized NoC is interactive and by clicking on one of its components, the user can find more detailed efficiency and performance information about it. The third and final feature is a detailed graphical analysis view in which the user can plot detailed component-level results such as those we presented in Chapter 5.

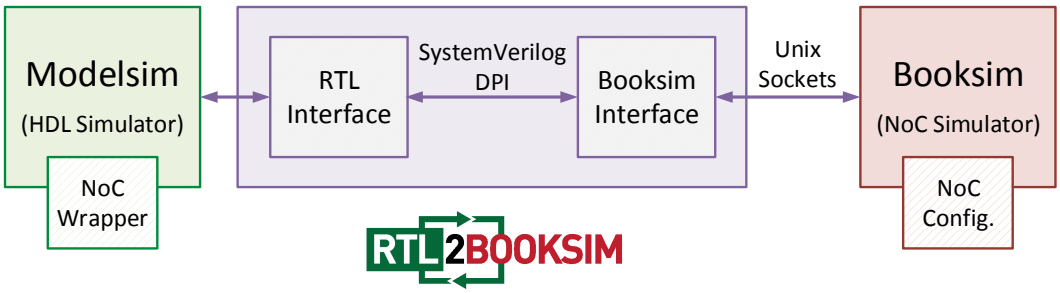


Figure 10.2: `RTL2Booksim` allows the cycle-accurate simulation of an NoC within an RTL simulation in Modelsim.

10.2 `RTL2Booksim`

In evaluating a system-level interconnect, such as an NoC, it is very important to measure its performance in terms of latency and throughput. To do that accurately, we need to perform cycle-accurate simulations of hardware designs that are connected to an NoC.

The conventional way to do this entails register-transfer level (RTL) hardware simulations using a simulator like Mentor Graphics' Modelsim. Designs are entered to these simulators using a HDL such as Verilog, SystemVerilog or VHDL. Therefore, we would need an HDL version of our NoC and FabricPort to properly simulate an embedded NoC. Furthermore, the HDL implementation of the NoC would have to be parameterizable (to be able to try out different NoCs) and fully verified (to avoid errors). We first tried borrowing such an open-source implementation [54], but we quickly found it very hard to use. Other than the intermittent bugs that we discovered, it was very challenging to properly match the interface of the NoC with our FabricPort. This is because every time we changed an NoC parameter, the packet format became noticeably different. The lack of documentation was also a barrier to the ease-of-use of this HDL NoC implementation.

Our second approach culminated in the creation of `RTL2Booksim`. Instead of using an HDL implementation of the NoC, we instead used a cycle-accurate software simulator of NoCs called Booksim [80]. This is advantageous because Booksim provides the same cycle-accuracy of simulation, but runs faster than an HDL model of the NoC, and supports more NoC variations. Additionally, we are able to define our own packet format (see Figure 9.3) which greatly simplifies the interface to the NoC. Finally, it is much easier to extend Booksim with a new router architecture or special feature, making it a useful research tool in fine-tuning the NoC architecture.

Booksim simulates an NoC from a trace file which describes the movement of packets in an NoC. However, our FabricPort and application case studies are still written in Verilog (an HDL). How do we connect our hardware Verilog components to a software simulator such as Booksim? This is the main purpose of `RTL2Booksim` – to interface HDL designs to Booksim. Figure 10.2 shows some details of this interface. The Booksim Interface[2] is able to send/receive flits and credits to/from the NoC modeled by the Booksim simulator. This Booksim Interface communicates with the Booksim simulator through Unix sockets. Next, there is an RTL Interface that communicates with our RTL HDL design modules. The RTL Interface communicates with the Booksim Interface through a feature of SystemVerilog called

[2]The Booksim Interface was initially developed by Robert Hesse in Prof. Natalie Enright Jerger's research group and was generously shared with us. We modified it to send and receive data at the flit granularity instead of the packet granularity and we added support for transmitting credits from the NoC to handle backpressure outside of Booksim.

the direct programming interface (DPI), which allows one to call software functions written in C/C++ from within a SystemVerilog design file. Through these two interfaces – the Booksim Interface and the RTL interface – we can connect any hardware design to any NoC that is modeled by Booksim.

```
topology = mesh;                // NoC topology
n = 2;                          // number of dimensions in topology
k = 4;                          // number of routers in each dimension
flit_width = 150;               // flit width
num_vcs = 2;                    // number of virtual channels
vc_buf_size = 10;               // buffer depth per VC
routing_function = dim_order; // routing algorithm
...
```

Listing 10.1: Sample Booksim NoC configuration file. Note that many more parameters are customizable in Booksim [80].

As Figure 10.2 shows, we can configure the NoC using a configuration file. Listing 10.1 shows an example of that file and highlights that it is very easy to change the NoC parameters. We made sure to have a simple NoC wrapper so that it acts exactly like an HDL NoC in an RTL simulation – snippets of the NoC wrapper are shown in Listing 10.2. `RTL2Booksim` is released as open-source and available for download at: `http://www.eecg.utoronto.ca/~mohamed/rtl2booksim`. The release includes push-button scripts that correctly start and end simulation for example designs using Modelsim and `RTL2Booksim`.

```
noc_wrapper #(
  .WIDTH_NOC   (150),  // channel width
  .N            (16),  // number of routers
  .NUM_VC        (2),  // number of virtual channels
  .DEPTH_PER_VC (10),  // buffer depth per VC
) noc_inst (
  .clk_noc (...),  // NoC clock
  .rst     (...),  // NoC reset
  // router 5 input
  .r5_data_in  (...),  // data
  .r5_valid_in (...),  // valid
  .r5_ready_out(...),  // ready
  // router 5 output
  .r5_data_out (...),  // data
  .r5_valid_out(...),  // valid
  .r5_ready_in (...),  // ready
  // other routers
);
```

Listing 10.2: The SystemVerilog component presented by `RTL2Booksim`. The designer simply includes this component in their RTL design and communication to Booksim is managed automatically

10.3 Physical NoC Emulation

To model the physical design repercussions (placement, routing, critical path delay) of using an embedded NoC, we emulate the existence of hardNoC routers on FPGAs. Using the floorplan produced by `NoC Designer`, we create a design partition for each router using Altera's Quartus II software. Each of the router partitions we create is essentially a locked region on the FPGA chip to which design modules can connect. By connecting the NoC-communicating modules of a design to these router partitions we can quantify some physical design metrics and answer related research questions:

- Routing Congestion: Each router in our proposed NoC contains 600 inputs and 600 outputs through the FabricPort. Additionally, the router's area is quite small since it is implemented in hard logic. What happens when a design module connects to this router using the soft FPGA interconnect? Does it produce routing congestion?
- Critical timing paths: Does the concentrated utilization of wires around a hard router perimeter force connections in a design to use more FPGA wires? Does this increase the length of a critical timing path?
- Area: By having a synthesis, placement and routing flow that includes an NoC, we can accurately measure the area of a design implemented on our FPGA.

To create a partition that accurately models a router, we must ensure the following steps are taken in creating and compiling the design in Quartus II:

1. Router partitions must have a register on the path from a router input to a router output. This is to break the timing path just as it would be broken with an actual embedded router that has several pipeline registers and buffers.
2. To prevent the router from being synthesized away if it is unused, we place a "synthesis noprune" pragma on its inputs and outputs.
3. First compile the dummy router partitions without the application you want to test. This is necessary so that the dummy router ports do not get optimized based on the application, rather, they are always fixed to this first empty compile.
4. Router partitions must use "post place and route" netlists so the router pins cannot be moved by Quartus placement to best match a design, as this would not occur with a real hard block.
5. Use "LogicLock" to fix routers to their locations in the FPGA fabric.
6. Connect your application modules to the emulated NoC in the design HDL code.
7. All unused router I/Os must terminate at a register and then a (virtual) I/O. This means, if you use just the input port of router 2, then you have to connect the outputs of router 2 through registers to a (virtual) output. This is to ensure that timing analysis doesn't ignore these paths.
8. All top-level modules connected to the NoC must have a "synthesis noprune" pragma on them if they have no path from a chip-level input to output. This will happen with modules that are only connected to a router.

Figure 10.3 shows a sample floorplan of our embedded NoC (from Chapter 7) on a Stratix V FPGA. The "router partitions" are *empty* partitions with 600 inputs and 600 outputs and a register between the inputs and outputs as shown in Listing 10.3. In the shown figure, the NoC is only connected to virtual I/Os around the router – virtual I/Os are LABs that are treated as I/Os by Quartus II.

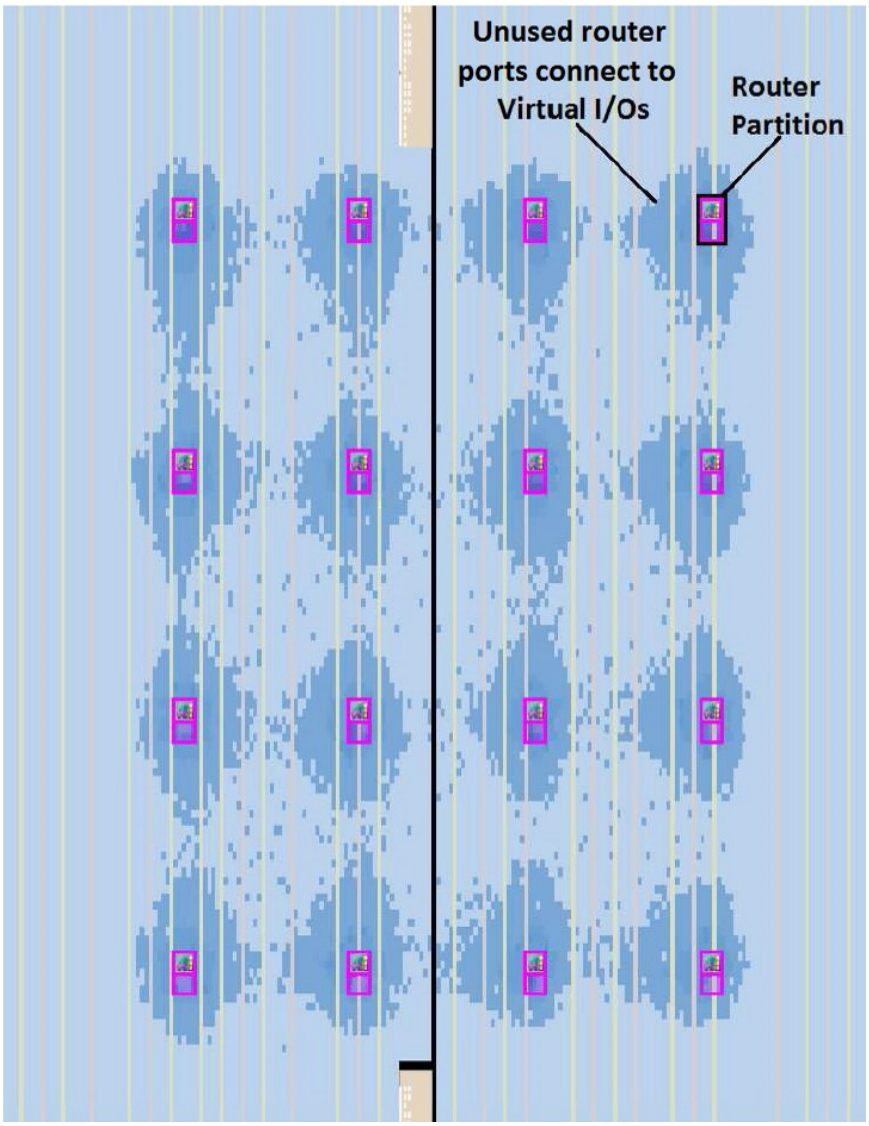


Figure 10.3: Sample floorplan of an emulated embedded NoC on a Stratix V 5SGSED8K1F40C2.

```
module router_partition #(parameter WIDTH = 600)
(
  input                    clk,
  input       [WIDTH-1:0] inputs /* synthesis noprune  */,
  output reg  [WIDTH-1:0] outputs /* synthesis noprune  */
);

always @ (posedge clk)
  outputs[WIDTH-1:0] <= inputs[WIDTH-1:0];
endmodule
```

Listing 10.3: Verilog code for a router partition to emulate an embedded NoC.

Chapter 11

Application Case Studies

Contents

In this chapter, we use the developed NoC architecture and FabricPort interface, and our simulation and prototying tools, to study three important applications. In the first case study we show that an embedded NoC can be used to distribute data from external memory throughout the FPGA with a much lower design effort than soft buses. We also show that an embedded NoC exceeds the efficiency of soft buses for most system sizes. Next, we show how a latency-sensitive image compression algorithm could benefit from our embedded NoC in terms of both improved frequency and reduced interconnect utilization. Finally, we show how an embedded NoC can be used to implement an Ethernet Switch and achieve 5× more switching than previously demonstrated on FPGAs.

11.1 External DDR3 Memory

FPGA designers currently use soft buses to integrate large systems; either by manual design or with system integration tools such as Altera's Qsys or Xilinx' XPS. The resulting hierarchical bus consists primarily of wide pipelined multiplexers configured out of FPGA logic blocks and soft interconnect. We propose changing the FPGA architecture to include an embedded hard NoC instead. Consequently, we must compare the efficiency of the soft bus-based interconnect to our proposed embedded NoC. Previous NoC versus bus comparisons have shown that NoCs become the more efficient interconnect only for large systems [152]; however, because we compare a *hard* NoC to a *soft* bus we find the NoC exceeds bus efficiency even for small systems. We also *measure* the design effort to close timing on soft buses by exposing the number of steps required to implement a soft bus using commercial tools – we believe that

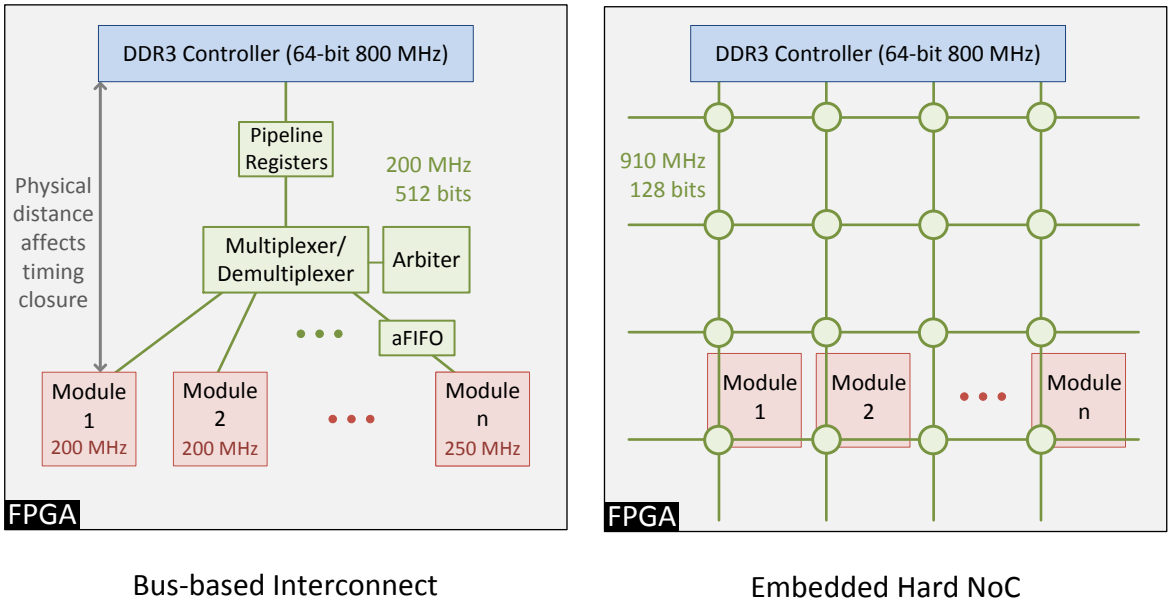


Figure 11.1: Connecting multiple modules to DDR3 memory using bus-based interconnect or the proposed embedded NoC. We use random traffic generators for these modules to emulate an application's compute kernels or on-chip memory.

any design effort for timing closure to external interfaces can be greatly reduced or eliminated when using an embedded NoC. Note that we do not discuss latency here as we do a much more detailed analysis of the latency of transaction systems in Chapter 13.

Figure 11.1 shows our experimental setup. We compare the efficiency of the application-tailored soft buses generated by Qsys (Altera's system integration tool) to our embedded NoC, when used to connect to external memory. By varying the number of modules accessing memory, we emulate different applications, and study interconnects of different sizes and interconnection capabilities For instance, packet processing applications [81] usually have only two memory masters, while video applications [16] can range from 6-18. We also vary the physical distance between the traffic generators and the memory they are trying to access; from "close" together to "far" on opposite ends of the chip. There are two common reasons whereby system-level interconnect spans a large distance. Either the FPGA is full and its modules are scattered across the chip, or I/Os on opposite ends of the chip are connected to one another. This physical remoteness makes it more difficult to generate the Qsys bus-based interconnect that ties everything together.

11.1.1 Design Effort

To highlight the designer effort necessary to implement different bus-based systems, Table 11.1 lists the steps taken to meet timing constraints using Qsys. A small system, with 3 modules located physically close to external memory, does not meet timing with default settings. Typically, designers would first switch on all optimization options in the CAD tools, causing a major increase in compile time to improve timing closure. Turning on extra optimization improved design frequency, but still fell short of the target by 6 MHz. We then enabled pipelining at various points in the bus-based interconnect between modules

and external memory. Only with 3 pipeline stages did we meet timing, and this came at a 45% area penalty and 340% power increase. Placing the modules physically farther away from memory makes the bus physically larger and therefore more difficult to design and less efficient: more than 2× the area and power. Furthermore, larger systems with 9 modules are still more difficult and time-consuming to design even with sophisticated tools like Qsys. Even after enabling the maximum interconnect pipelining, we need to identify the critical path and manually insert an additional stage of pipeline registers to reach 200 MHz. This timing closure process is largely ad-hoc, and relies on trial-and-error and designer experience. Our proposition eliminates this time-consuming process; an embedded NoC is predesigned to distribute data over the entire FPGA chip while meeting the timing and bandwidth requirements of interfaces like DDR3.

Table 11.1: Design steps and interconnect overhead of different systems implemented with Qsys.

System Size	Physical Proximity	Design Effort	Frequency		Area	Power
Small (3 modules)	close	– –	✗	187 MHz	95 LABs	17 mW
		Max tool effort (+ physical synthesis)	✗	194 MHz	95 LABs	17 mW
		Auto interconnect pipelining (3 stages)	✓	227 MHz	139 LABs	75 mW
	far	– –	✗	92 MHz	96 LABs	48 mW
		Max tool effort (+ physical synthesis)	✗	96 MHz	96 LABs	51 mW
		Auto interconnect pipelining (4 stages)	✓	205 MHz	299 LABs	199 mW
Large (9 modules)	far	– –	✗	70 MHz	249 LABs	49 mW
		Max tool effort (+ physical synthesis)	✗	70 MHz	250 LABs	49 mW
		Auto interconnect pipelining (4 stages)	✗	198 MHz	757 LABs	302 mW
		Manual interconnect pipelining (1 stage)	✓	216 MHz	801 LABs	307 mW

11.1.2 Area

Figure 11.2 shows how the area and power of bus-based interconnects and embedded NoCs vary with the number of modules. We compare a Qsys-generated 512-bit soft bus to a 16-node 128-bit hard NoC embedded onto the FPGA as shown in Figure 11.1. Chapter 4 details the methodology for finding the area/speed/power of hard NoCs; we follow the same methodology and build on it in this section. For instance, we compute hard NoC power for our benchmarks by first simulating the benchmarks to find the total data bandwidth flowing through the NoC; we then multiply the total bandwidth by our "Power per BW" metric that we computed for hard NoCs [4].

The plotted area of the hard NoC is independent of the design size up to 16 nodes as we must prefabricate the entire NoC onto the FPGA chip and always pay its complete area cost; nevertheless, its entire area is less than optimized bus-based interconnect for designs with more than 3 modules accessing DDR3 memory. On the other hand, buses that are generated by Qsys become larger and more difficult to design with more connected modules, consuming up to 6% of the logic blocks on large FPGAs to connect to a single external memory. It is possible to combine bus-based and NoC-based communication in one system, maintaining the high configurability crucial to FPGAs. For example, if more than 16 modules access an external memory, two modules may be combined using a small soft bus to share one NoC fabric port. Additionally, portions of a design with low bandwidth may also choose to use a bus for communication.

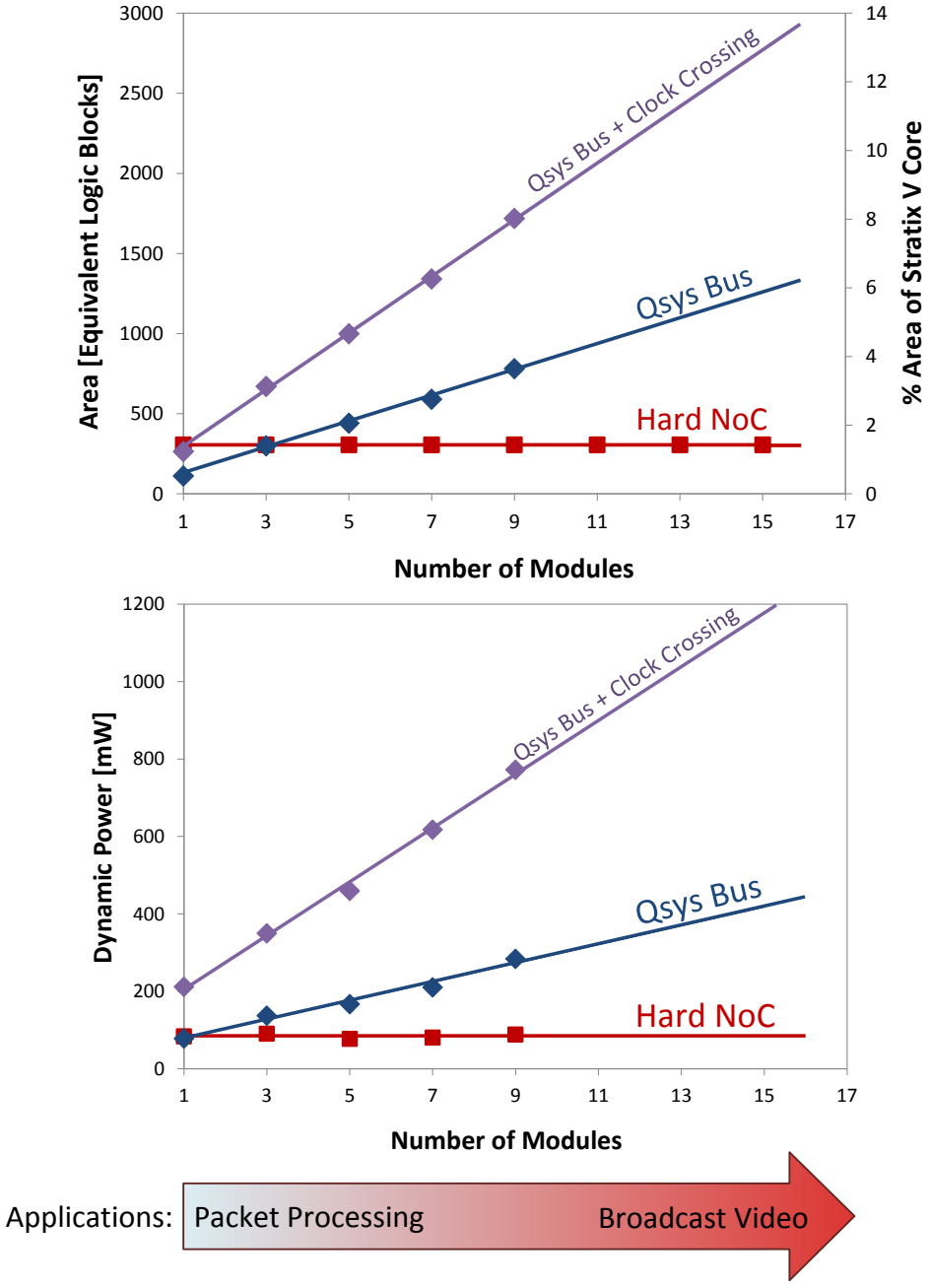


Figure 11.2: Comparison of area and power of Qsys bus-based interconnect and embedded NoC with a varying number of modules accessing external memory. Example applications corresponding to the number of modules are annotated.

The embedded NoC includes clock-crossing and data-width adapters at its FabricPorts allowing us to connect modules running at any independent clock frequency. This is common in commercial designs as IP modules may have any frequency within the limits of the FPGA, rather than matching the DDR3 fabric interface frequency of 200 MHz. Figure 11.2 plots the area of Qsys buses when the connected

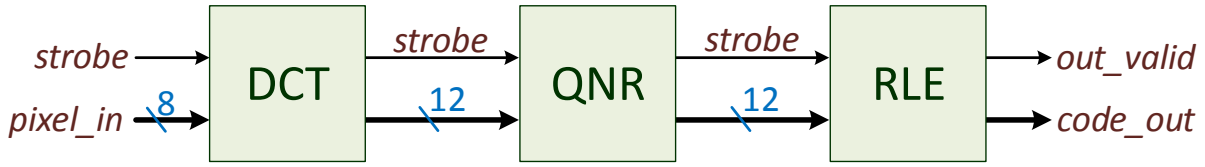


Figure 11.3: Single-stream JPEG block diagram.

modules operate at 250 MHz and thus require aFIFOs to connect to the bus. These aFIFOs are as wide as the bus (512 bits) and hence incur a large area overhead; in fact, the area required by a one-module Qsys system with clock crossing is approximately equal to the complete hard NoC area.

Each link of the hard NoC can carry more than the entire bandwidth of an 800 MHz 64-bit DDR3 channel; the NoC is underutilized in our study. Consequently, it can support connectivity to multiple I/O interfaces at once with no additional area; this further widens its efficiency advantage over buses for more complex systems. For instance, an FPGA in a recent Maxeler system connects to 6 DDR3 controllers.

11.1.3 Dynamic Power

Figure 11.2 shows that NoC Dynamic power is also somewhat constant vs. the number of modules accessing one DDR3 interface; this is for two reasons. First, we always move the same bandwidth on the NoC divided equally amongst all accessing modules, so the interconnect itself always transports the same total bandwidth. Second, we average the power for our systems when their modules are placed physically "close" and "far" from the DDR3 interface; meaning that the same bandwidth moves a similar average distance for all systems. We use the same assumptions for Qsys buses but they become more power hungry with larger systems. This is because the bus itself uses more resources – such as wider multiplexers and more soft interconnect – when spanning a larger distance to connect to a higher number of modules. Furthermore, we increasingly need to add power-hungry pipeline registers for larger Qsys buses to meet the 200 MHz timing requirement. In contrast, the embedded NoC routers are unchanged when we increase the number of modules. As Figure 11.2 shows, the embedded NoC is consistently more power efficient than tailored soft buses. The power gap widens further when we include clock-crossing FIFOs in a Qsys-generated bus, whereas an embedded NoC already performs clock-crossing in its fabric ports.

11.2 Parallel JPEG Compression

We focused on the advantages of embedded NoCs in easing design effort and improving efficiency in the DDR3 interface case study. In this section, we investigate a streaming image compression application that is very well-suited for FPGAs. We evaluate the timing variability of such an application when it is subject to different I/O placement constraints, and we quantify its utilization of soft interconnect resources and compare it to our NoC-based implementation.

We use a streaming JPEG compression design from [73]. The application consists of three modules as shown in Figure 11.3; discrete cosine transform (DCT), quantizer (QNR) and run-length encoding (RLE). The single pipeline shown in Figure 11.3 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 [73]. The components of this system are therefore latency-sensitive as they rely on pixels arriving every cycle, and the modules do not respond to backpressure.

We parallelize this application by instantiating multiple (10–40) JPEG pipelines in parallel; which means that the connection width between the DCT, QNR and RLE modules varies between 130 bits and 520 bits. Parallel JPEG compression is an important data-center application as multiple images are often required to be compressed at multiple resolutions before being stored in data-center disk drives; this forms an important part of 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 Figure 9.7. 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.

11.2.1 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 Figure 11.6 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. Figure 11.4 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.

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 Figure 11.6 – 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 Figure 11.4. 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.

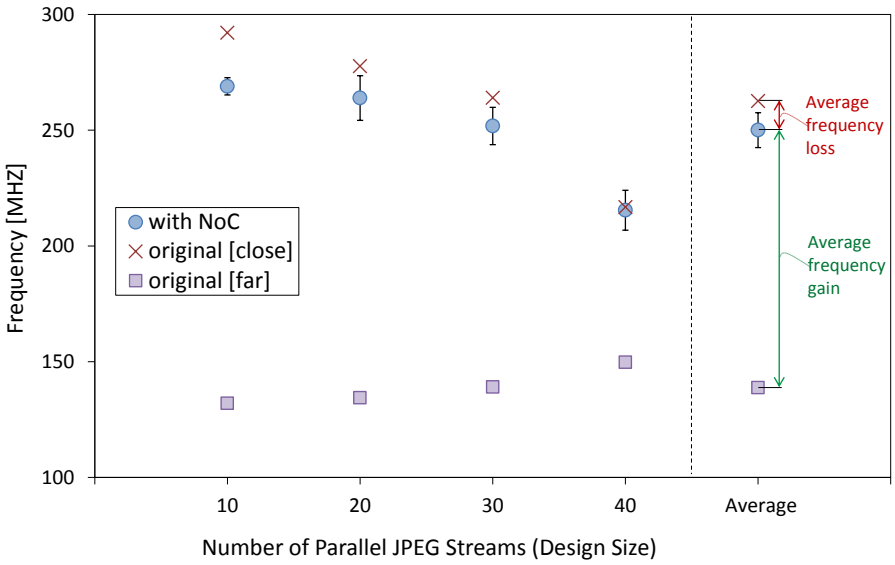


Figure 11.4: Frequency of the parallel JPEG compression application with and without an NoC. The plot "with NoC" is averaged for the two cases when it's "close" and "far" with the standard deviation plotted as error bars. Results are averaged over 3 seeds.

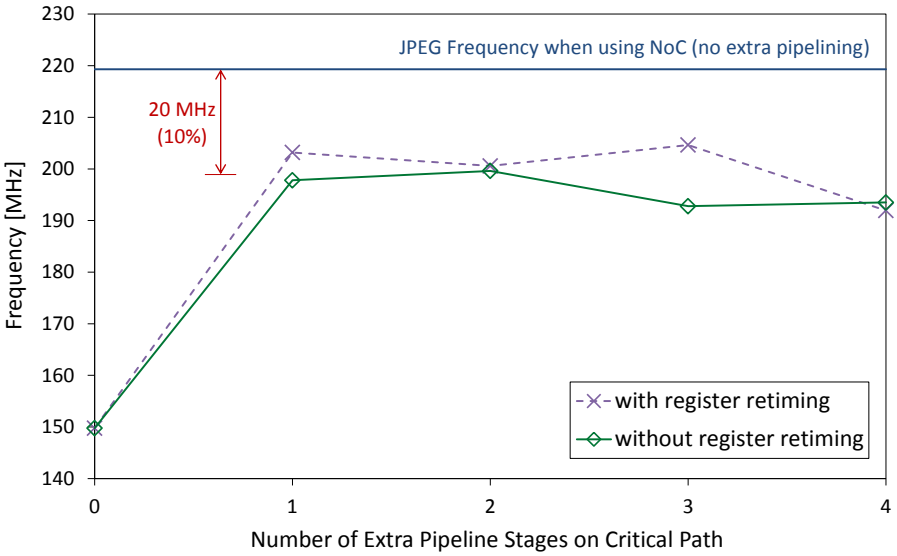


Figure 11.5: Frequency of parallel JPEG with 40 streams when we add 1-4 pipeline stages on the critical path. Frequency of the same application when connected to the NoC is plotted for comparison. Results are averaged over 3 seeds.

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 [115]. 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 Figure 11.5.

Table 11.2: 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)	+13.2%	+10.2%
	Horizontal (R3,R6)	+7.8%	
Long	Vertical (C14)	-47.2%	-38.6%
	Horizontal (R24)	-31.6%	

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

The plot shows two things. First, the frequency of the pipelined version never becomes as good as that of the NoC version even with 4 pipeline stages on the critical path – on average, there is a 10% difference in frequency. Secondly, it doesn't really matter how many pipeline registers we place on the critical path, nor does it matter much whether register retiming is enabled. This is because register retiming occurs before placement and routing in the CAD flow, and therefore has no physical awareness on where the register will actually be placed on the FPGA device.

11.2.2 Interconnect Utilization

Table 11.2 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.

Figure 11.6 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 Figure 11.6.

11.3 Ethernet Switch

In this application case study [1], we demonstrate the built-in switching and buffering capability of an embedded NoC. We show that the embedded NoC is not only a means of data transport, but is also a very fast and efficient buffered crossbar.

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 [31]. 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 [50].

[1] Most of this case study was done by Master's graduate Andrew Bitar [21]. My contribution was in the experimental inception of the plot in Figure 11.9, and in connecting the Ethernet switch through the FabricPort and `RTL2Booksim`.

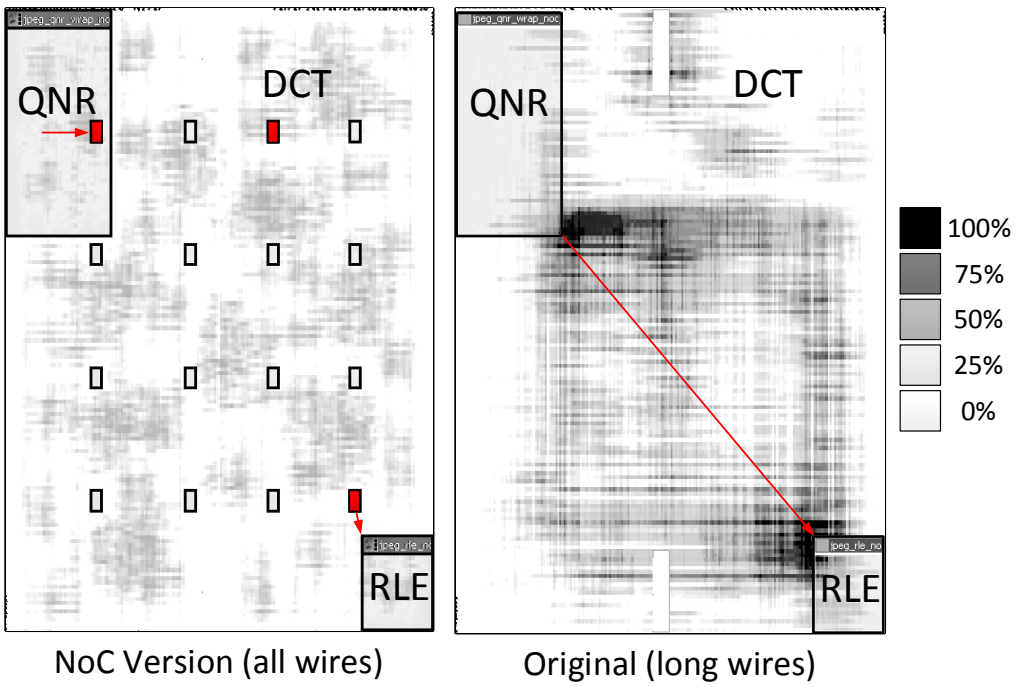


Figure 11.6: Heat map showing total wire utilization for the NoC version, and only long-wire utilization for the original version of the JPEG application with 40 streams when modules are spaced out in the "far" configuration. In hot spots, utilization of scarce long wires in the original version goes up to 100%, while total wire utilization never exceeds 40% for the NoC version.

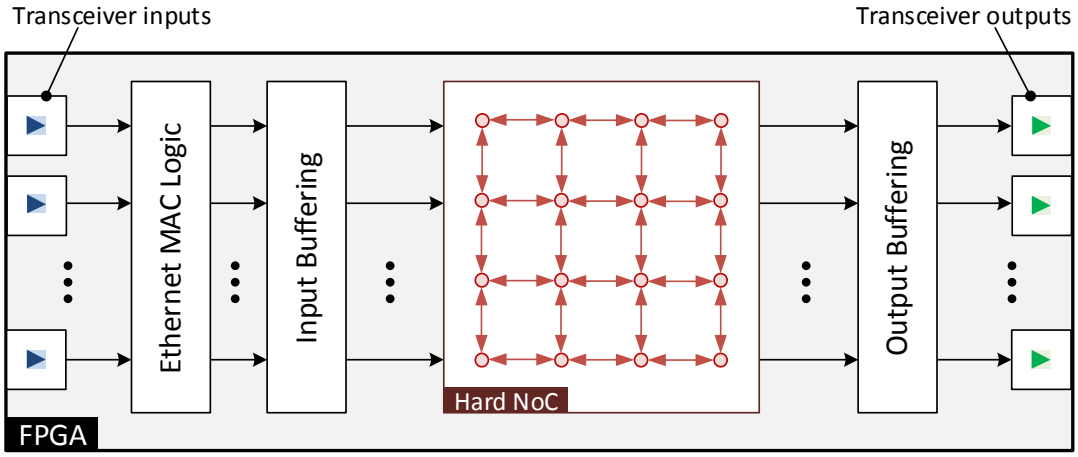


Figure 11.7: Block diagram of an Ethernet switch that uses a hard NoC for switching and buffering.

An Ethernet switch is a large buffered crossbar for switching Ethernet frames from a transceiver input to a transceiver output. It is one of the most important building blocks of communication networks but has largely been dominated by ASIC or custom implementations because of its high bandwidth demands. An Ethernet switch is latency-insensitive as data is already grouped into Ethernet packets that can take a variable number of cycles to cross the FPGA. It is an interesting form of streaming application: while the data is in the form of streams, the streams are being switched between multiple destinations.

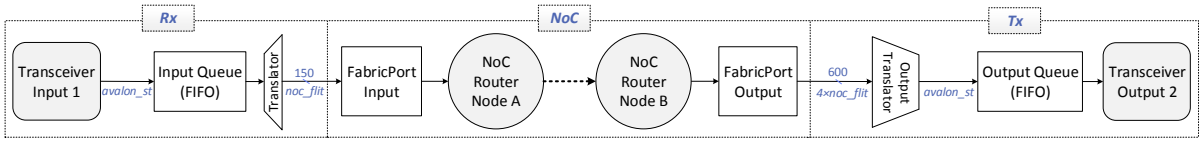


Figure 11.8: Functional block diagram of one path through our NoC Ethernet switch [11].

Table 11.3: 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**

Interestingly, FPGAs have a great deal of serial transceiver I/O bandwidth (nearly 3 Tb/s in the largest Virtex Ultrascale device [146]) but FPGAs are inefficient in implementing large crossbars with centralized arbitration, making them unable to effectively switch all of the I/O bandwidth they can send or receive. The highest-bandwidth Ethernet switch demonstrated on FPGAs has supported 160 Gb/s [50] while ASIC implementations have exceeded 3 Tb/s [39].

We use an embedded NoC in place of the switch crossbar to provide a more scalable, high-bandwidth yet efficient solution for Ethernet switching – Figure 11.7 shows a block diagram of such an implementation. Sixteen transceiver inputs and outputs are connected to the 16 routers in the hard NoC after going through media access control (MAC) logic and buffering as shown in Figure 11.7. By leveraging the embedded NoC for switching and buffering, our NoC-based switch (soft logic plus hard NoC) area is ~3× smaller the best previously published implementation [50]. As described below, our switch also has more than 5× the switching bandwidth of [50], which means that it is over 15× more efficient at switching per unit area than a traditional FPGA. Finally, we consider the design of the NoC-based switch to be simpler than the alternative: timing closure is easier as the most complex (switching) functionality is done in the pre-fabricated NoC, and it is not necessary to use the clever but more complex hardware design techniques employed in [50].

Figure 11.8 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 11.3. 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 and a breakdown of its FPGA resource utilization is shown in Table 11.3. Because we take advantage of the NoC's switching and buffering our switch is ~3× more area efficient than previous FPGA Ethernet switches [50], and 15x more efficient in area per unit of bandwidth.

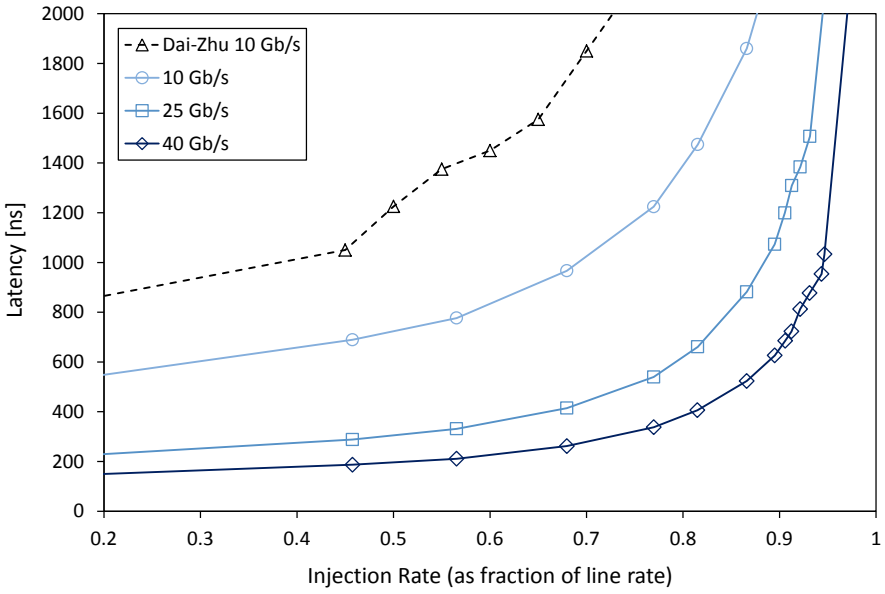


Figure 11.9: Latency vs. injection rate of the NoC-based Ethernet switch design given line rates of 10, 25, and 40 Gb/s [11], and compared to the Dai/Zhu 16×16 10 Gb/s FPGA switch fabric design [50].

Two important performance metrics for Ethernet switches are bandwidth and latency [58]. The bandwidth of our NoC-based Ethernet switch is limited by the supported bandwidth of the embedded NoC. Each NoC link has a bandwidth capacity of 22.5 GB/s (180 Gb/s) and 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 [31]. While previous work on FPGA switch design has achieved up to 160 Gb/s of aggregate bandwidth [50], our switch design can achieve up to 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. With changes only to the soft logic of the design we can also support alternative switches such as a 32-port 10 Gb/s Ethernet switch; we are not limited to a 16-port switch simply because we have a 16-node NoC. Fig. 11.9 plots the latency of our Ethernet switch at its supported line rates of 10 Gb/s, 25 Gb/s and 40 Gb/s. No matter what the injection bandwidth, the NoC-based switch considerably outperforms the traditional FPGA switch [50] 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.

Part III

Computer-Aided Design

Table of Contents

Chapter 12

LYNX CAD System

Contents

In Part II we made hard NoCs usable by enforcing specific design constraints and using the FabricPort. We tried to make embedded NoCs very easy to use by simplifying its interface; for instance, the only backpressure signals we use are "ready" and "valid" (instead of the more complicated credit-based flow control). However, to fully leverage the NoC, an application designer still needs some expert knowledge. For example, what is the best mapping of design modules to NoC routers? Which VC should I use? When do I use combine-data mode? How do I set the control bits of a head flit in a packet? etcetera. In this chapter, we present a CAD tool –LYNX – that can automatically connect *any* user design using an embedded NoC with *any* parameters. In this chapter, we outline the different steps in the LYNX CAD flow, and how we implement them.

LYNX is CAD tool that automatically connects an FPGA application using an NoC. LYNX connects application modules to NoC routers, generates soft-logic wrappers that are required for semantically correct and high-performance communication, sets the constant control bits in a data packet, and generates a Verilog model of the complete system. 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.

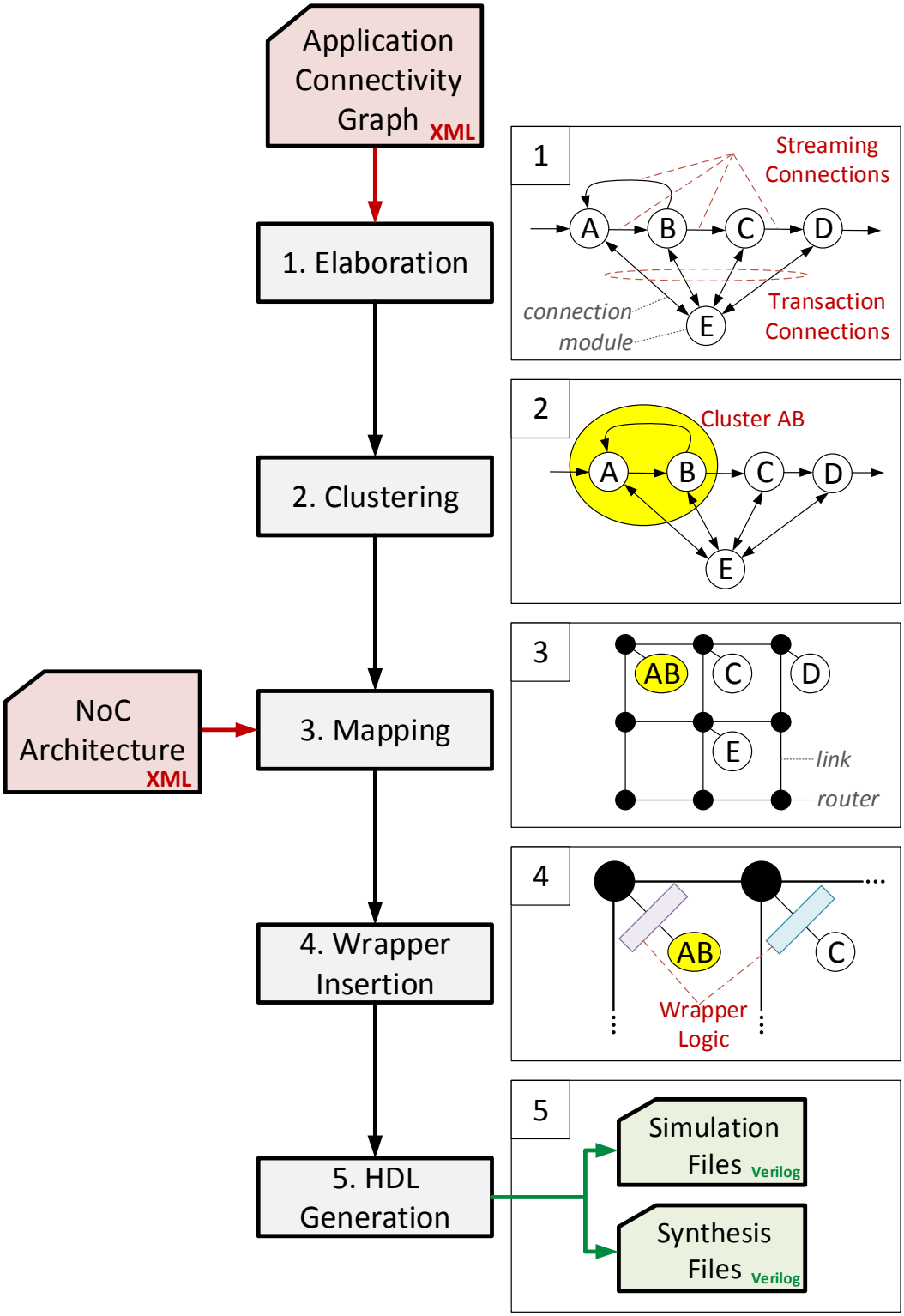


Figure 12.1: Overview of the LYNX CAD flow for embedded NoCs.

Figure 12.1 shows an overview of the LYNX CAD flow. An application is entered as a LYNXML[1] file like the example shown in Listing 12.1. 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 [37]. 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 manage traffic as we discuss in Chapter 13. Finally, simulation and synthesis files are generated to be able to use LYNX results with traditional synthesis and simulation tools.

```
<design name="p2p_example">

 <!-- Modules -->
 <module name="src1">
   <bundle name="obun" direction="output">
    <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" direction="input">
   <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"/>

</design>
```

Listing 12.1: Sample LYNXML description of two modules connected by a 128-bit wide connection.

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

- 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.

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

12.1 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. Streaming connections refer to unidirectional communication between modules, while transaction connections consist of both requests and replies. Elaboration also groups transaction connections to-and-from the same modules together into a "connection group" as shown in the illustration of Figure 12.1, where modules A, B, C, D and E form one connection group. We create this grouping because the number of modules in a transaction system affects later steps in the CAD flow as we discuss in Chapter 13.

Programmatically, the elaboration step consists of a linear inspection of connections, and then when a transaction connection is identified, the design graph is traversed to group all related transaction connections together. Elaboration therefore runs very quickly (typically below 1 ms) with a worst-case complexity of $O(n)$, where n is the number of connections.

12.2 Clustering

A major limitation of NoCs is that their latency cannot be reduced beyond the latency required to traverse 2 FabricPorts and 1 router as shown in Figure 9.1 – in our embedded NoC, this latency is approximately 8 clock cycles. Consider an application that contains a feedback loop like that illustrated in Figure 12.1. 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 [37]. The authors use "Tarjan's algorithm" [140] 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. We also use Tarjan's algorithm to cluster modules that are involved in a feedback connection; however, we ignore transaction connections 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 modules in a feedback cycle and directly connects ports in this cycle using soft point-to-point links.

12.3 Mapping

Mapping is the core algorithm of the LYNX flow – it connects application modules to NoC routers. As shown in Figure 12.1, mapping takes an NoC architecture file as input. By changing NoC parameters, we can optimize a soft NoC for an application. In the case of architecting an embedded NoC before FPGA

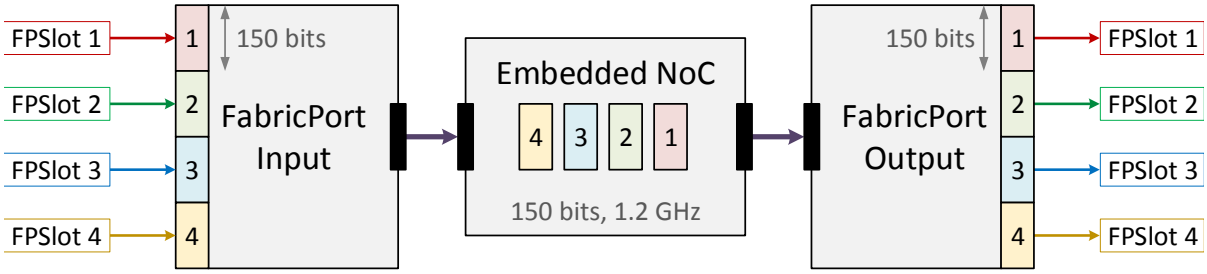


Figure 12.2: A FabricPort time-multiplexes wide data from the FPGA fabric to the narrower/faster embedded NoC. 1–4 different bundles can connect to the shown FabricPort by using one-or-more FabricPort Slots (FPSlots) depending on the width and number of VCs.

manufacture, the system architect can use LYNX to try out different NoCs for important application benchmarks before deciding on the final architecture.

12.3.1 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 for transport [11]. Figure 12.2 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, so ensuring that each bundle uses a different VC effectively decouples the bundles completely so that if two bundles are connected at a FabricPort output and one of them stalls, the other can continue to receive data on a different VC. This also ensures deadlock freedom [11] as described in Chapter 8. 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 12.1 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.

12.3.2 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 12.3.1

LYNX uses simulated annealing to map an application to an NoC. Prior work has shown that many optimization algorithms can suit the mapping problem [127]; however, we use simulated annealing be-

Table 12.1: 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.

cause of its flexibility and scalability to larger systems. Additionally, simulated annealing makes it easy to change the cost function or add legality constraints without much effort. 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. 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 connections 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 12.1 is the simulated annealing cost function used in mapping. W_{1-6} are constant weights[2] 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.

[2]Note that the final values for the cost function constants in Equation 12.1 are not included in this thesis as they are not yet fully determined. A larger benchmark set and further experiments are required before reasonable cost function constants are decided. The current values for $W_1 - W_6$ are 0.25, 2, 1, 10000, 2000 and 1. Off-NoC connections and modules connecting to multiple routers are heavily penalized and link overutilization is emphasized more than latency.

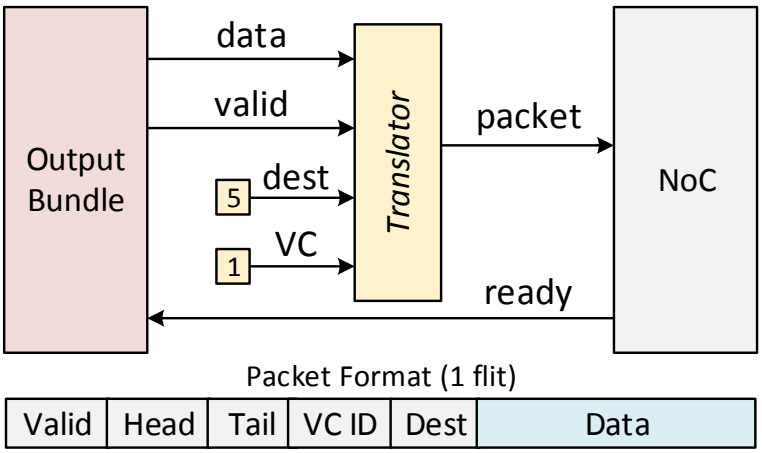


Figure 12.3: The simplest translator takes data/valid bits and produces an NoC packet. LYNX determines the destination/VC bits statically if the sending output bundle has only one possible dest/VC as shown (destination = 5 and VC = 1); otherwise, the application logic has to set the dest/VC for each outgoing data.

$$
\begin{aligned}
Cost \;=\; & \sum_{L_i \in Links} W_1\big(Util(L_i)\big)^{W_2} \\
& + \sum_{C_i \in Conns} \Big(W_3 Latency(C_i) + W_4 OffNoC(C_i) \Big) \\
& + \sum_{M_i \in Modules} W_5\big(Routers(M_i)\big)^{W_6}
\end{aligned}
\tag{12.1}
$$

12.4 Wrapper Insertion

"Wrappers" encompass any soft logic required to make communication on the NoC possible and performant. 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 Figure 12.3. 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 in a packet.

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.

12.5 HDL Generation

The 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. This methodology is described in Chapter 10.

12.5.1 Mimic Flow: Simulation and Synthesis

In the beginning of this chapter, 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 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. Additionally, these dummy modules also produce a trace file of all the packet transfers. The traffic generators can be parametrized to send data every n cycles, where n can be set by the user. Additionally, modules that both receive and send data can be configured to respect data dependency so that a module only sends data once all or some of its inputs have received data. We use these dummy "mimic" modules to simulate an application's connectivity together with an embedded NoC or bus interconnect. The simulation results (trace files) identify the latency and throughput at each point of an ACG and can be used to evaluate the performance of the used interconnect, be it an NoC or a bus.

In MimicSyn, we generate synthesis files using dummy modules that are heavily pipelined so as not to limit frequency. They contain a mix of logic, RAM and arithmetic where the ratio of logic/RAM/arithmetic is tunable by the user through parameters to better model the actual application being evaluated. The dummy modules are connected to the interconnect – NoC or bus – which is then synthesized using standard FPGA CAD tools like Quartus II. The synthesis results provide a reasonable estimate of frequency and identifies whether the interconnect contributes to limiting overall system frequency.

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 application module implementation. This would better allow the fast investigation of different system-level interconnects without a set of complete applications as a prerequisite.

Chapter 13

Transaction Communication

Contents

Communication in FPGA applications can be classified into two main types: streaming or transaction (sometimes referred to as "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 chapter, 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 priorities and change arbitration shares, and the role of transaction ordering and its high-performance implementation. Most of the techniques we

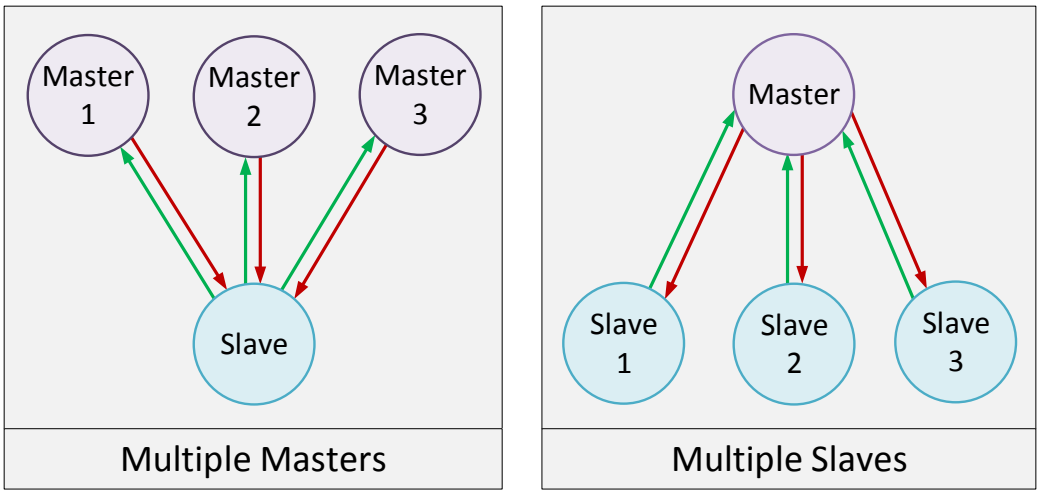


Figure 13.1: Transaction systems building blocks.

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`+ 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 – this is the same NoC we proposed in Chapter 7 except that we increase the number of VCs to 4 for maximum flexibility in managing transaction traffic.

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 Figure 13.1. After presenting the transaction system components, we discuss multiple-master and multiple-slave systems in Sections 13.2 and 13.3 respectively. The methods we present with each type of system are composable and can easily be used together in multiple-master multiple-slave systems.

13.1 Transaction System Components in NoCs

Figure 13.2 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 issued if a "traffic manager" allows it. We discuss traffic managers in detail in the following sections. A 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.

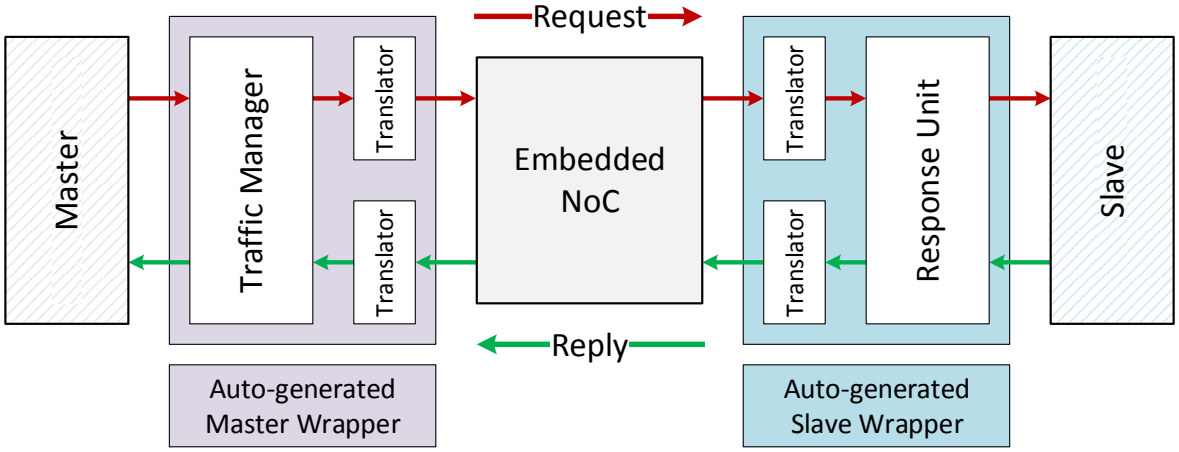


Figure 13.2: System-level view of master-slave connections using the NoC.

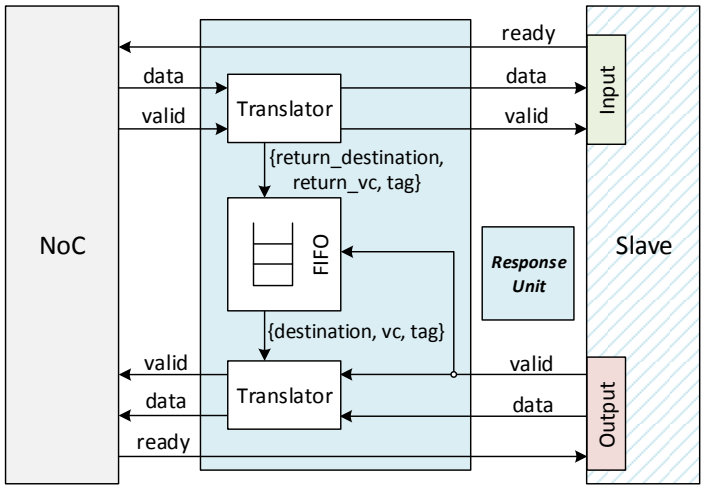


Figure 13.3: A response unit at a slave buffers the return address information (return destination router and VC) and optionally a tag, and attaches it to the slave response.

13.1.1 Response Unit

Figure 13.3 shows an implementation for the response unit. A simple translator first inspects the master request and extracts the data, valid, return destination, return VC and tag (optional). Data and valid are forwarded to the slave module, while the return information (destination router, VC, tag[1]) 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 ever overflow. 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.

[1] A tag is an optional field to uniquely identify a request or a reply.

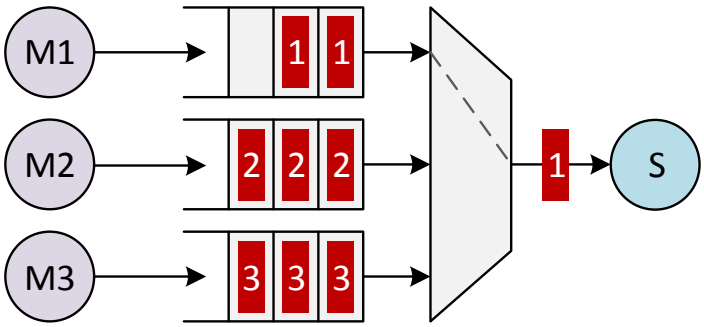


Figure 13.4: Traffic build-up in a multiple-master system. Requests can build-up quickly in a multiple-master interconnect without traffic management, because all masters can send requests at the same time to a shared slave (which can only process 1 request at a time).

13.2 Multiple-Master Systems

Multiple-master systems are very common in FPGA designs; an example is 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 assign different priorities to different masters sharing the same slave. In this section, we'll look at systems that have multiple masters and a single slave.

13.2.1 Traffic Build Up (in NoCs)

Before discussing our implementation, we present an important problem that exists in any pipelined multiple-master interconnect; Figure 13.4 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 in NoCs which are a shared interconnect resource. When the buffers start becoming full, the latency of *all* packets that are using the same routers (even if they are not going to the congested slave module) increases very quickly. To mitigate this problem, we introduce traffic management schemes that avoid traffic build-up altogether.

13.2.2 Credit-based Traffic Management

One way to solve the traffic build up problem is to employ a credit-based traffic management scheme. Figure 13.5 shows the Credits Traffic Manager, which is placed at each master to limit the number of outstanding requests. The counter in Figure 13.5 is set to a selected "number of credits", whenever the

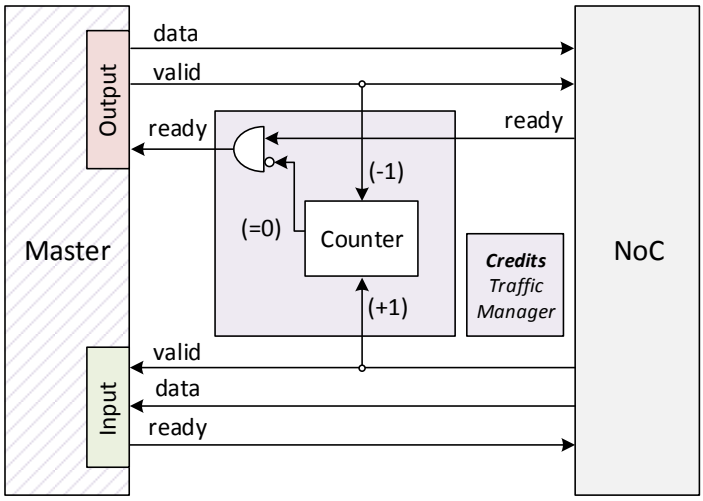


Figure 13.5: Credits traffic manager to limit traffic between a master sharing a slave with other masters.

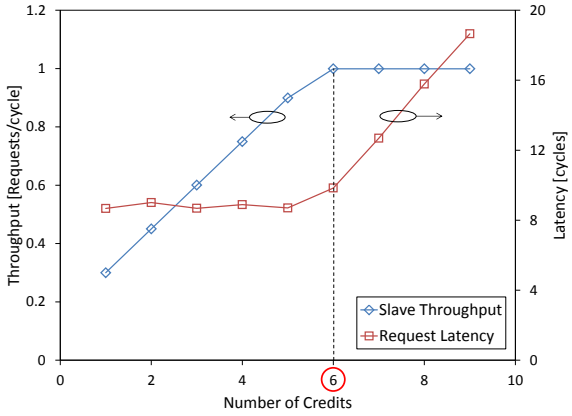
master sends a request the credits are decremented by 1, and whenever a reply is received the credits increase by 1. Zero credits stalls the master and this ensures that no new requests are made until replies for the outstanding requests are received.

It is crucial to select the number of credits appropriately – too many credits and traffic will build up, too few credits and the slave will be underutilized. To better visualize this, see Figure 13.6: 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 and then it starts worsening latency beyond that. The ideal number of credits at each master (circled in Figure 13.6) depends on the number of masters and the round-trip latency (between sending a request and receiving its reply), and follows the following equation:

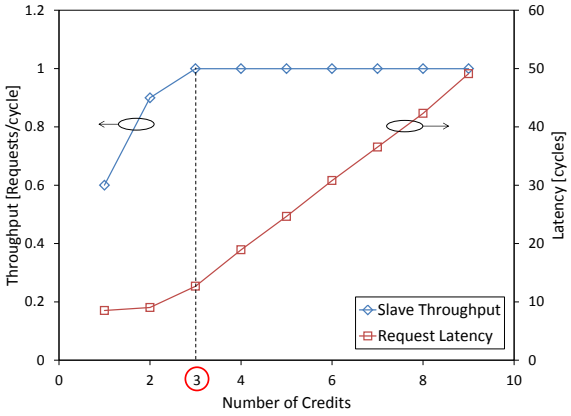
$$\text{Credits}_{\text{ideal}} = \frac{\text{Latency}_{\text{roundtrip}}}{\text{Number of Masters}} \tag{13.1}$$

To understand why equation 13.1 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 13.1 effectively shares that slave bandwidth equally by dividing the $\text{Latency}_{\text{roundtrip}}$ by the number of masters. Note that the average $\text{Latency}_{\text{roundtrip}}$ can be used in the case that the $\text{Latency}_{\text{roundtrip}}$ is different for each master; however, latency is typically very close for different NoC locations (as Figure 9.1 shows) so the resulting unfairness is likely very small.

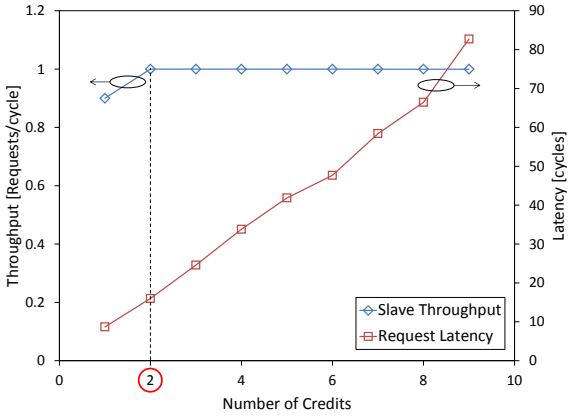
Figure 13.7 plots the ideal number of credits for multiple-master systems while varying the number of masters. The "simulation" data series in this plot was experimentally determined by trying out different number of credits. Traffic generators attempt to send data every cycle but are stalled if the number of credits is zero, or if backpressure is received from the NoC. Equation 13.1 is also plotted (dotted line) – the model agrees with our simulation results very closely and the discrepancies only exist



(a) 3 masters.



(b) 6 masters.



(c) 9 masters.

Figure 13.6: Investigation of the ideal number of credits for multiple-master communication with 3, 6 and 9 masters.

because our Credits Traffic Managers only support an integer number of credits. We include this model in `LYNX` which automatically instantiates the traffic manager and sets the correct number of credits for any multiple-master systems.

13.2.3 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 "without clock crossing" in Figure 13.8, 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.

Figure 13.8 compares the roundtrip latency of multiple-master systems of different sizes. The latency of Qsys buses increases linearly with the number of masters because of the traffic build-up problem

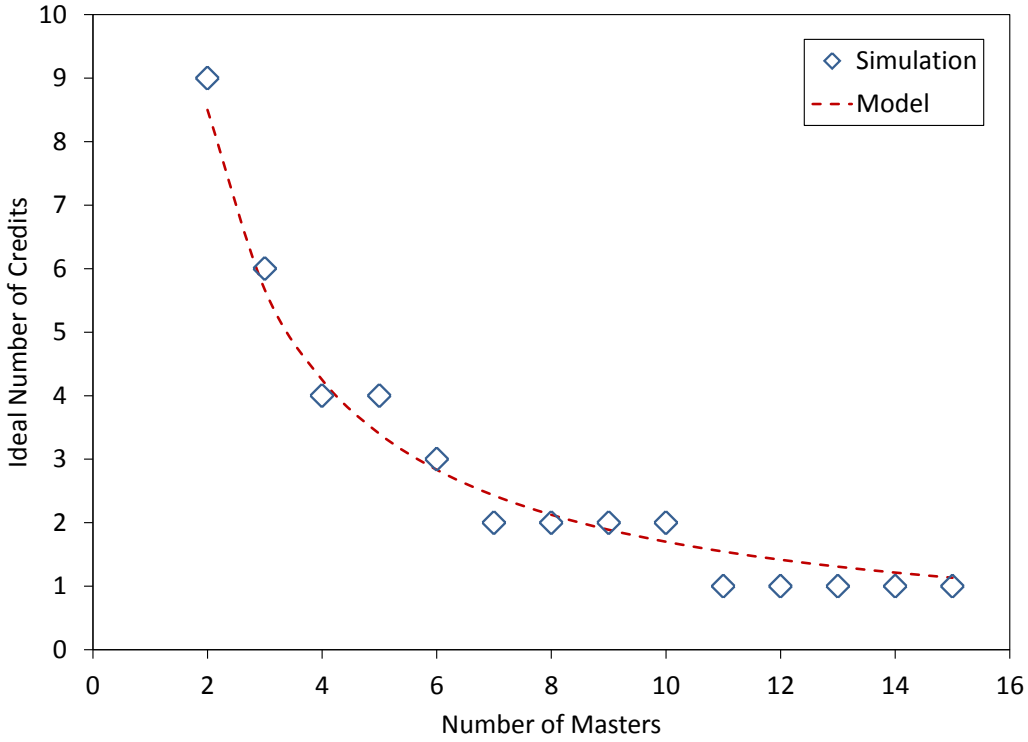


Figure 13.7: Ideal number of credits for NoC traffic managers to minimize request latency. Both the experimental evaluation from Figure 13.6, and the model from Equation 13.1 are shown.

discussed in Section 13.2.1. However, when traffic managers are used in `LYNX`, the latency remains more-or-less constant[2] 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.

13.2.4 Priorities and Arbitration Shares

Qsys instantiates a fair round-robin arbiter by default. However, the user can also assign "arbitration shares" for each master specifying exactly how many requests to accept from each master in each round of arbitration thus giving a higher priority to masters with more shares [49]. This unfair arbitration is easily implemented in the Qsys bus central arbiter. For NoCs: how do we implement these priorities where arbitration is not central in a single arbiter, but rather distributed among the routers in the NoC?

We build on our credit-based traffic management scheme to assign priorities. We assign more credits to high-priority masters, and fewer credits to lower-priority masters, such that the total number of credits are still equal to $Latency_{roundtrip}$ (or a value very close to it) to avoid traffic build-up. Figure 13.9 shows an experimental investigation of this priority scheme where we have 10 masters in the system, only one of which has a higher priority. The bars show the throughput of the priority master and normal master as we vary the credits ratio between them. When the credits ratio is 1, they both have the same number of credits and therefore the same throughput – one tenth. However, as we increase the credits ratio,

[2]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.

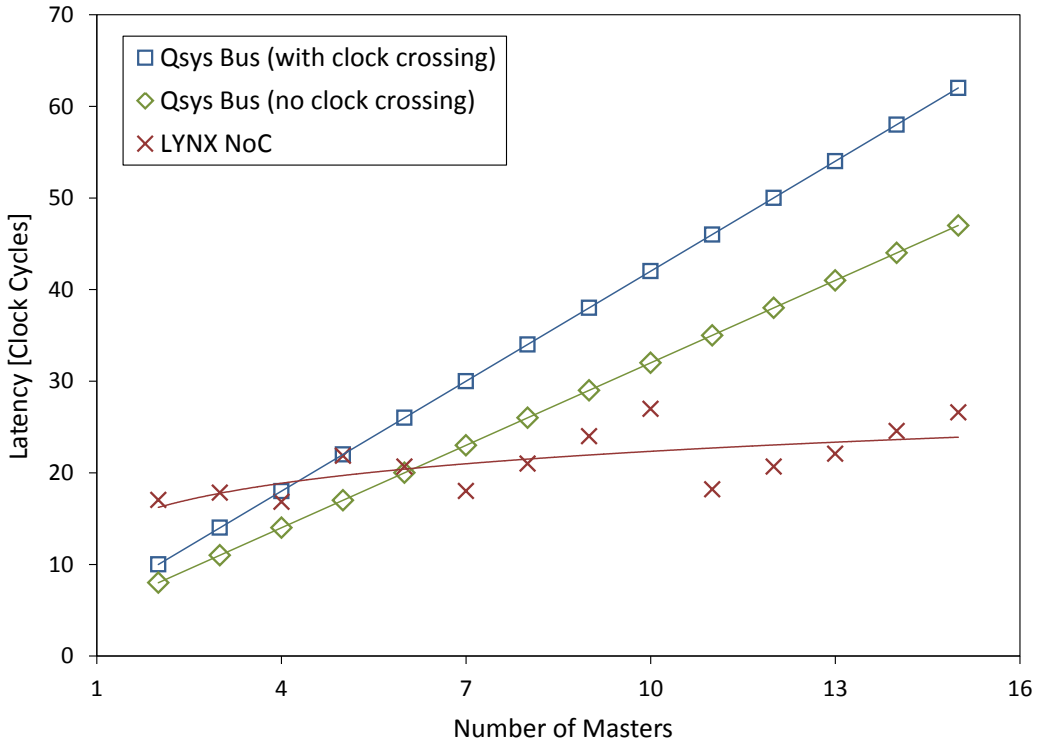


Figure 13.8: Comparison of Lynx NoC and Qsys bus latencies in a high-throughput system.

the priority master's throughput increases while the normal masters' throughput decreases. The line in Figure 13.9 plots the arbitration share of the priority master as it varies with the credits ratio, and it shows a linear relationship between the two. By adjusting the credits ratio at each master we can implement different arbitration shares in a distributed NoC interconnect similarly to Qsys centralized buses.

13.3 Multiple-Slave Systems

An example of a multiple-slave system is a processor (master) controlling multiple slaves such as memory units, accelerators 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 a high-performance multiple-slave using NoCs while maintaining proper transaction ordering to avoid data dependency hazards.

13.3.1 Ordering in Multiple-Slave Systems

Figure 13.10 shows the timing diagram of a master connected to two slaves. It sends request 1 to slave 1, and 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 can be problematic in an FPGA system where the master expects replies to arrive in

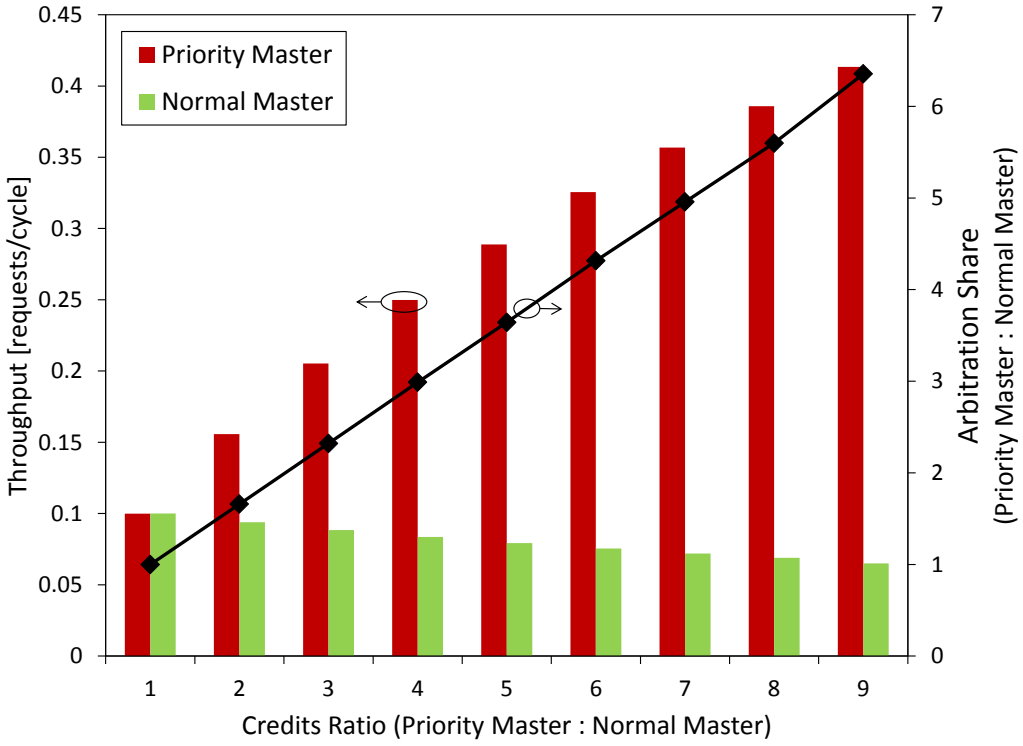


Figure 13.9: Changing the number of credits at a master increases its arbitration share thus giving it priority access to the shared slave. The plot shows a linear relationship between the credits ratio and the throughput share. Experiments run on a 10-master system.

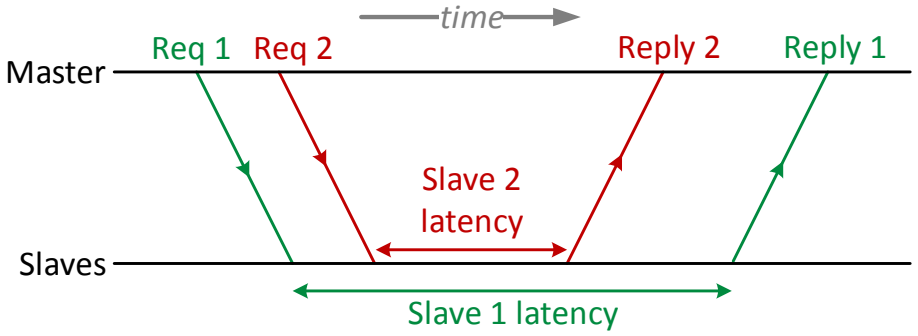
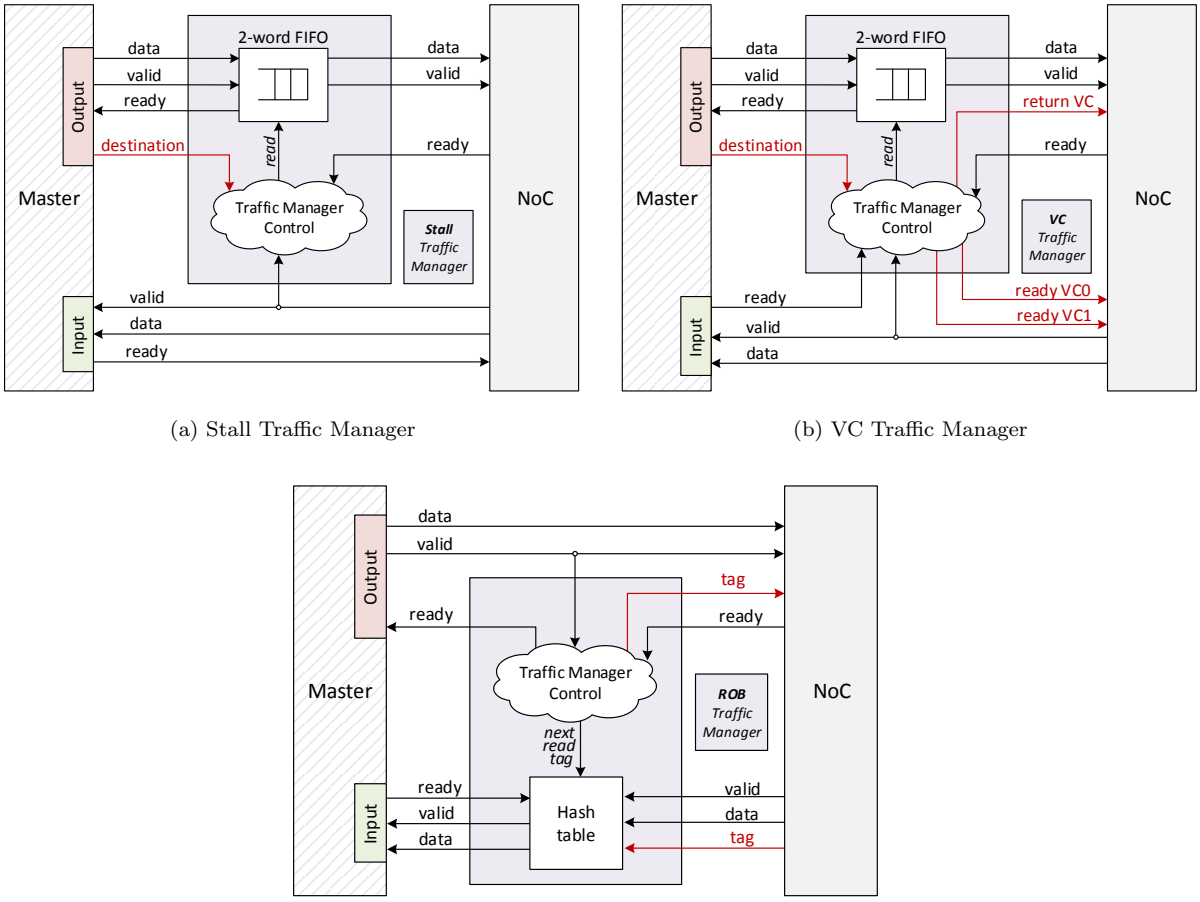


Figure 13.10: Requests to multiple slaves can result in out-of-order replies. Different slave processing latencies cause reply 2 to arrive at the master before reply 1.

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 guarantee ordering 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 sections.



(a) Stall Traffic Manager

(b) VC Traffic Manager

(c) ROB Traffic Manager

Figure 13.11: Different traffic managers to manage communication between a master and multiple slaves.

13.3.2 Three Traffic Managers for Multiple-Slave Systems

In this section we present three traffic managers to ensure ordering within multiple-slave systems. All three traffic managers include a Credits Traffic Manager (Section 13.2.2) 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. Figure 13.11 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.

$$\text{Throughput}_{Stall} = \frac{\text{N}_{\text{req1slave}}}{\text{N}_{\text{req1slave}} + \text{Latency}} \tag{13.2}$$

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 Figure 13.11 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).

$$\text{Throughput}_{\text{VC}} = \begin{cases} \text{N}_{\text{VC}}(\frac{\text{N}_{\text{req1slave}}}{\text{N}_{\text{req1slave}}+\text{Latency}}) & \text{N}_{\text{VC}} < \text{N}_{\text{slave}} \\ 1 & \text{N}_{\text{VC}} \geq \text{N}_{\text{slave}} \end{cases} \tag{13.3}$$

Reorder Buffer (ROB) Traffic Manager

The ROB Traffic manager adds an 8-bit number (or "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 13.2.2). The response unit (Figure 13.3) 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.

$$\text{Throughput}_{\text{ROB}} = \begin{cases} \frac{\text{N}_{\text{credits}}}{\text{Latency}} & \text{N}_{\text{credits}} < \text{Latency} \\ 1 & \text{N}_{\text{credits}} \geq \text{Latency} \end{cases} \tag{13.4}$$

13.3.3 Traffic Managers Performance and Efficiency

Figure 13.12 plots the master throughput in multiple-slave systems generated by `LYNX` and Qsys. On the x-axis of Figure 13.12, 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 Figure 13.12; however, if the number of slaves were 4 or less, the throughput would always be the maximum. Finally, an ROB Traffic Manager with $\text{N}_{\text{credits}} = \text{Latency}$ always has the maximum throughput. With fewer credits, the master throughput

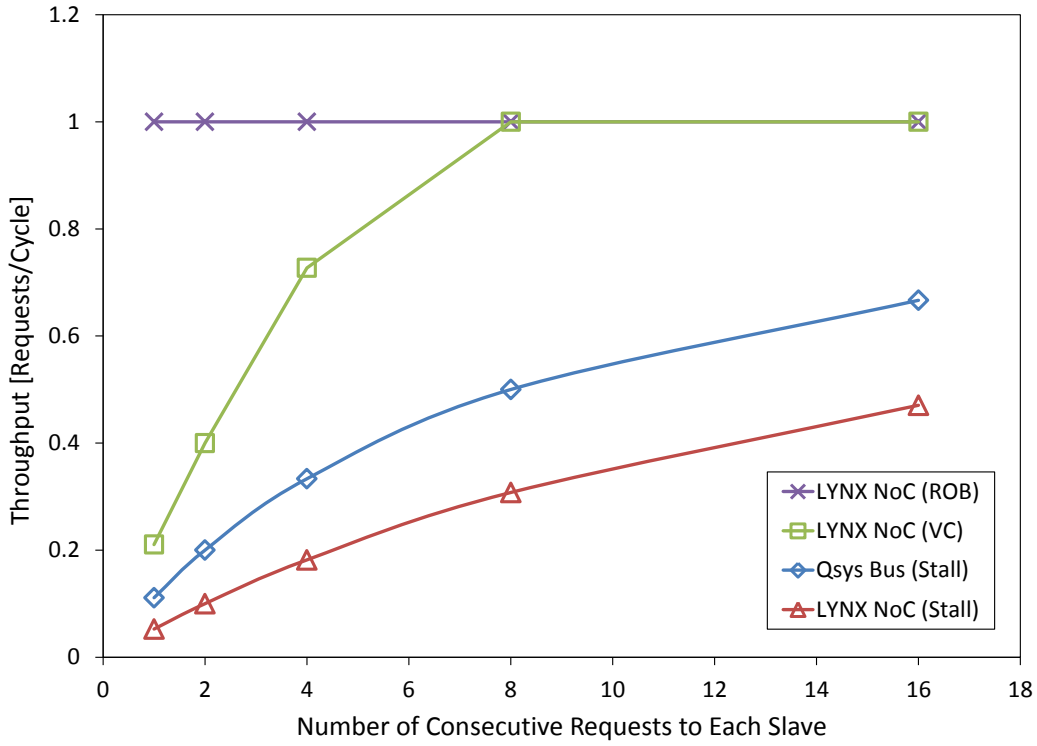


Figure 13.12: Maximum master throughput in a multiple-slave system with more than 4 slaves. The number of consecutive requests to the same slave is varied on the x-axis. Different traffic managers are analyzed.

decreases linearly until, with 1 credit, the ROB Traffic Manager becomes the same as the Stall Traffic Manager.

Figure 13.13 compares the area of the three traffic managers as we vary the number of credits. We measure area in *equivalent* Altera logic clusters (LABs[3]). The Stall and VC Traffic Managers use ~23 LABs, mainly for the 2-word FIFO which gets implemented using FPGA flip flops. The ROB Traffic Manager contain 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 and 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). Additionally traffic managers of all types (for both multiple master and multiple

[3]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 [124].

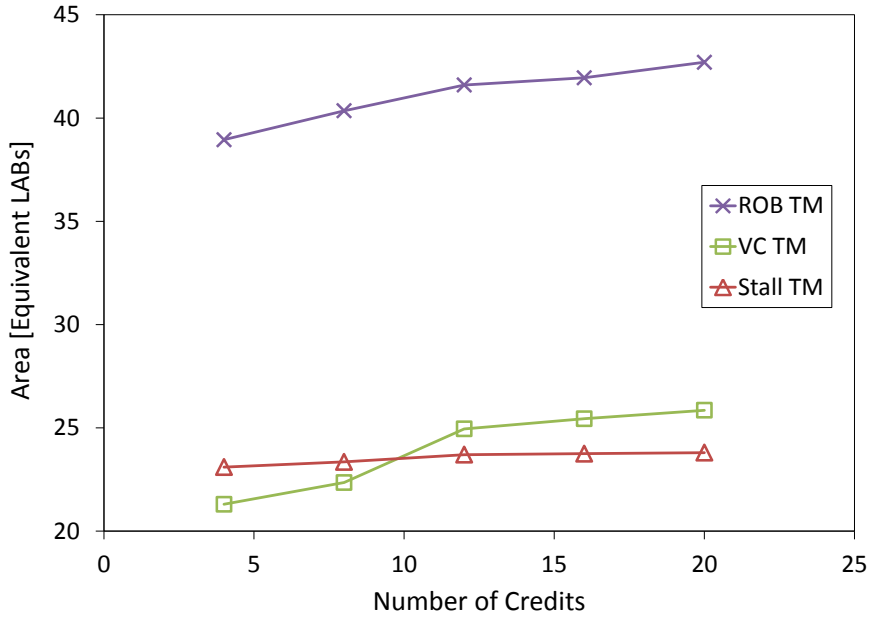


Figure 13.13: Area of the different traffic managers with width=300 bits as we increase the maximum number of outstanding requests (credits).

slave systems) are distributed, with one unit being placed near each master, therefore, traffic managers do not impact system scalability in terms of frequency or latency.

13.4 Limit Study

We have shown how `LYNX` automatically connects an application to an (embedded) NoC, and 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, priorities, 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 [13]. In different FabricPort modes (discussed in Section 12.3.1), we can connect up to 64 modules of 150-bit width, 32 Modules 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.

13.4.1 Area

Figure 13.14 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

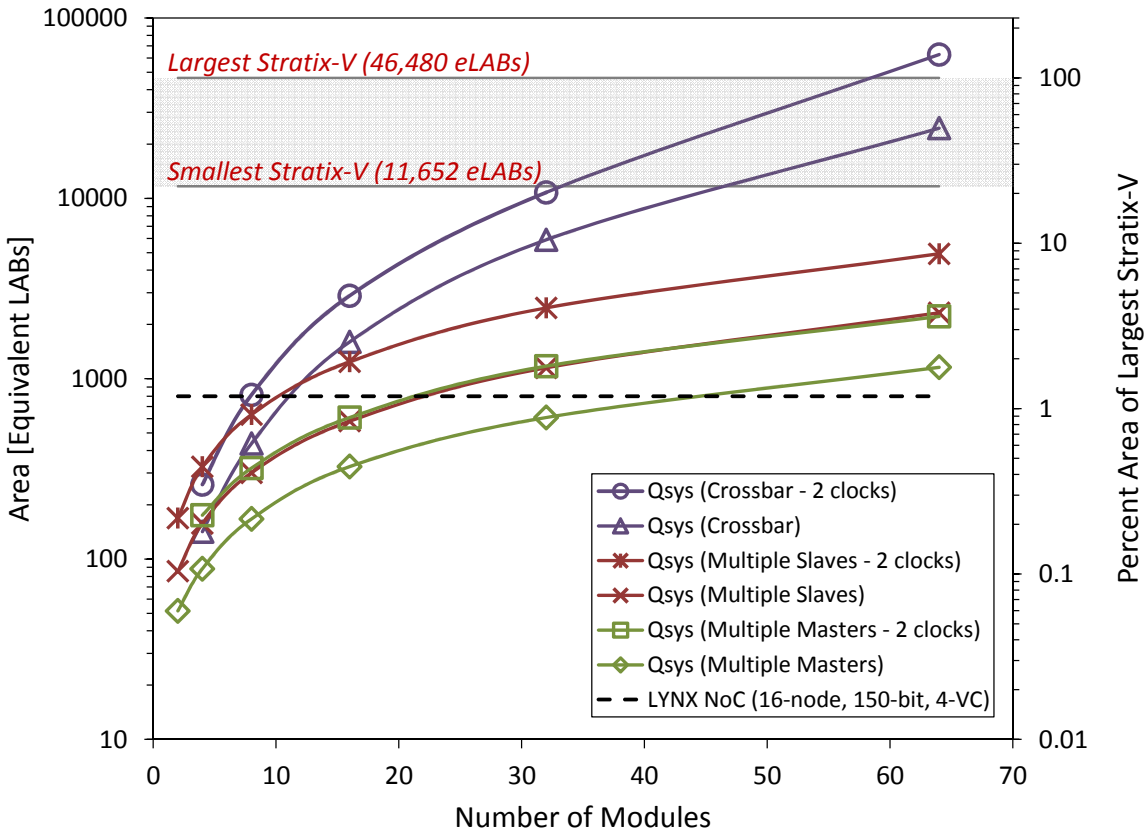


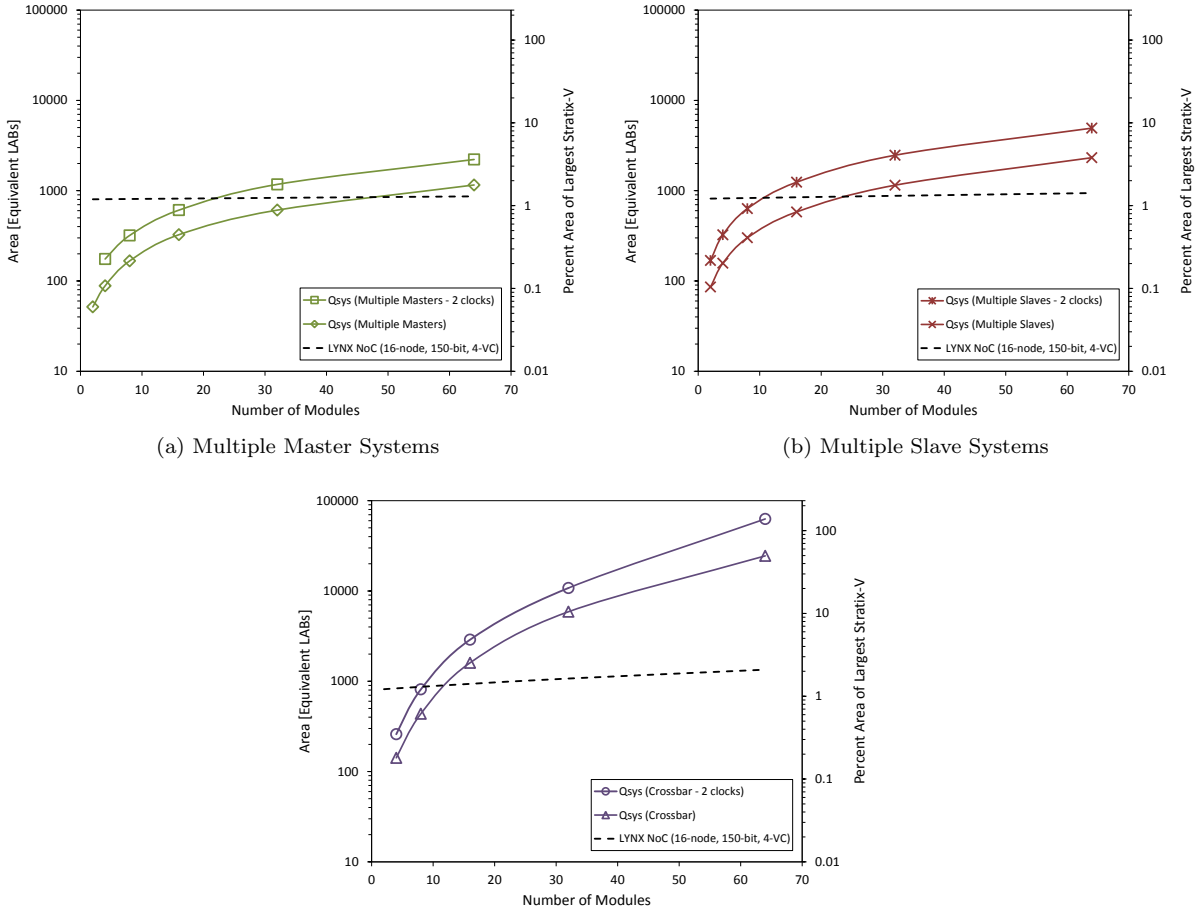
Figure 13.14: Area of Qsys buses of varying number of modules and 128-bit width. Different connectivity buses are shown: multiple master accessing a single slave, single master issuing requests to multiple slaves and a fully connected crossbar.

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 [11].

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 buses increase 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 infeasibility of large crossbars on FPGAs[4], and the efficiency of a hard system-level interconnect such as an embedded NoC.

Figure 13.14 only included the area of the embedded NoC and neglected the area overhead of wrappers. For completeness, Figure 13.15 plots the area of each type of system (multiple master, multiple slave or crossbar) separately, and adds the area of the NoC traffic managers and response units to the NoC area. The areas for the credits traffic manager and the response unit are each ~1 LAB, while the area of the stall traffic manager is ~15 LABs. For the multiple master system in Figure 13.15a there are 63 masters and 1 slave so we need 63 credits traffic managers and 1 response unit. For the multiple

[4]Large crossbars can push FPGAs into new markets such as building an ethernet switch [11, 31], or implementing hardware mapreduce [134].



(a) Multiple Master Systems

(b) Multiple Slave Systems

(c) Crossbar Systems

Figure 13.15: Breakdown of Figure 13.14 including the wrappers area for traffic managers and response units that are required with the hard NoC.

slave system in Figure 13.15b, we need 1 stall traffic manager for the master and 63 response units for the slaves. The added area due to the wrappers is almost negligible for both the multiple master and multiple slave systems; however, for the crossbar system, we use 32 stall traffic managers to ensure that transaction ordering is correct, which increases area proportionally to system size as Figure 13.15c shows. Nevertheless, the system size only grows from 800 LABs in a 1-master, 1-slave system to 1344 LABs in a 32-master, 32-slave system; we conclude that the wrappers area is relatively small for realistic systems.

13.4.2 Frequency

Figure 13.16 shows the frequency of transaction systems connected by Qsys buses and the `LYNX` embedded NoC. We use the "MimicSyn" flow (see Section 12.5.1) 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), multipliers (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.

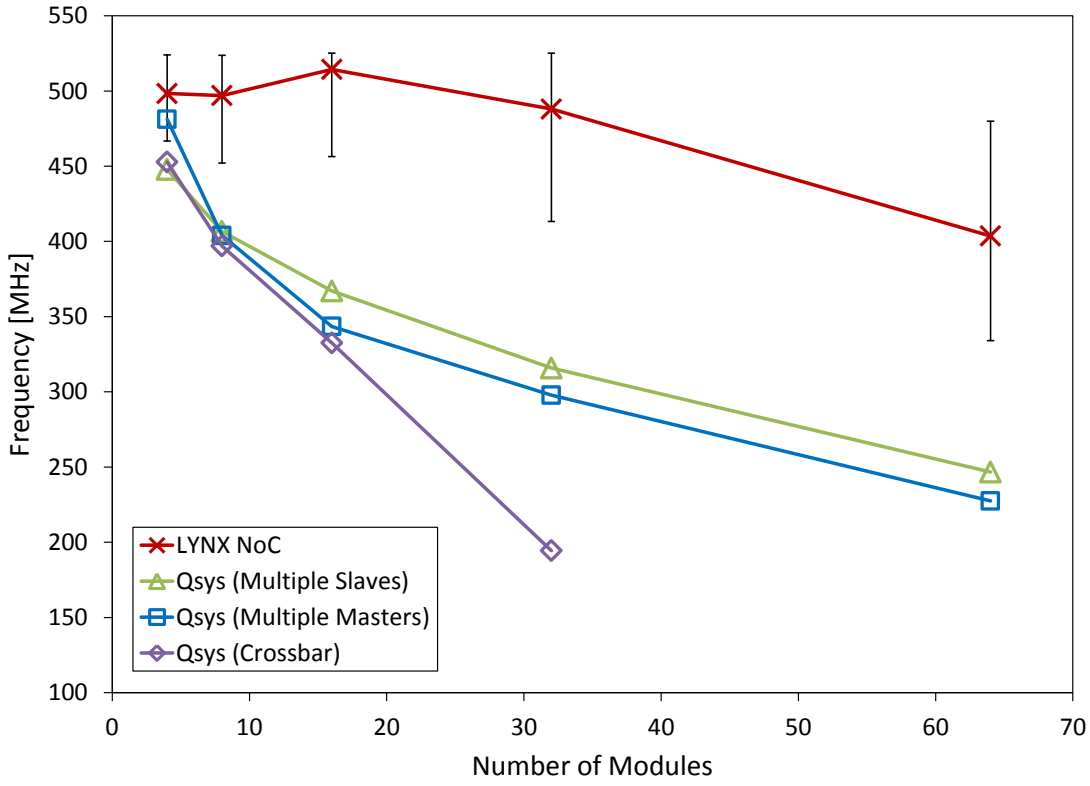


Figure 13.16: Comparison of NoC and bus frequency for 128-bit systems. When using the embedded NoC, each module can run on an independent clock; the error bars show the range of frequencies for each module.

When connected to the embedded NoC, each module in the system operates at an independent clock – we plot the minimum, maximum and average of these module clock frequencies using error bars on the `LYNX` data series in Figure 13.16. 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 [11]. Figure 13.16 shows that, compared to single-clock Qsys buses, the `LYNX` embedded NoC achieves between ~1.5× the frequency of soft buses. 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.

13.5 Transaction Systems Summary

This chapter focused on the implementation of transaction communication using an embedded NoC. We specified the components of a transaction system connected through an NoC; a traffic manager is required at the master side, while a response unit is required at the slave side. We then delved into the circuit details of each of those components and found that different traffic managers are required for different systems. We used a credits traffic manager to avoid traffic build-up in NoCs in a multiple-master system and create fair distributed arbitration. We also implemented unfair (priority) arbitration by adjusting the number of credits at each master. For multiple-slave systems we presented three traffic managers that ensure correct transaction ordering. By adding full transaction communication support to our `LYNX` CAD flow and embedded NoC interconnect, we have a complete solution that is directly comparable to current commercial system integration tools like Qsys. Our embedded NoC results outperform Alteras Qsys soft customized buses in latency, throughput and area efficiency, especially for larger and higher-throughput systems.

Chapter 14

Summary and Future Work

Contents

14.1 Summary

This thesis investigated the addition of embedded NoCs to FPGAs for system-level communication. In Part I we performed a detailed efficiency and performance analysis of NoC subcomponents. We measured the area, speed and power of the five router subcomponents and the NoC links when implemented either hard or soft. This quantified the design tradeoffs between hard and soft implementation of each NoC component and we used these component-level results to architect complete NoCs that are suitable for FPGAs. We presented both mixed (hard routers and soft links) and hard (hard routers and links) NoCs, with particular attention to their floorplan in and interface to the existing FPGA fabric. Our architecture exploration recommended the use of a hard NoC in FPGAs because of its efficiency and performance in transporting data. Additionally, a hard NoC is completely disjoint from the FPGA fabric making its speed and area known at design time – an important property that is not fulfilled by soft or mixed NoCs. We prototyped a hard 16-node NoC suitable for 28-nm FPGAs with the capability of transporting 150 Gb/s on each of its links, and we showed that it only consumes ~1.3% area of a modern FPGA.

After architecting the NoC and quantifying its area, speed and power in Part I, we defined how to design FPGA applications using the embedded NoC in Part II. We started by creating the FabricPort: a

flexible interface between NoC routers and the FPGA fabric. The FabricPort is responsible for adapting the width and frequency between the FPGA and the NoC – this allows us to connect modules of any width and frequency while running the NoC at a fixed (and very fast) speed to minimize communication latency and maximize throughput. We also discussed IOLinks which extend the NoC to connect directly to I/O interfaces, which improved the latency of accessing external memory in our case study. Next, we defined the rules and constraints that are required for semantically-correct FPGA communication using an embedded NoC. We discussed important constraints related to data ordering and deadlock-freedom, and we investigated the requirements of both latency-sensitive and latency-insensitive design.

This provided all the information we required to implement complete applications using our NoC; however, we still lacked the tools to implement an FPGA system that includes an embedded NoC. This motivated the creation of simulation and prototyping tools that emulate an embedded NoC within an FPGA. We created `NoC Designer` to estimate the area, speed and power of NoCs, and `RTL2Booksim` to simulate an embedded NoC including FabricPorts with an application. We also emulated the existence of an embedded NoC in Altera's Quartus II to investigate the physical design repercussions of using such an NoC. Using the tools we developed, we were able to implement complete applications using the embedded NoC, and we implemented three. We first showed that using a prefabricated NoC eased connection to external DDRx memory by eliminating time-consuming timing-closure iterations, and is also more efficient compared to a soft bus. The second application was parallel image compression and it highlighted the predictable speed of connecting to an NoC router from anywhere on the FPGA chip. In addition, we showed an embedded NoC improves soft interconnect utilization, and that there are no interconnect hot spots around NoC routers. The final application case study was an Ethernet switch – based on our embedded NoC – that supported 5× more switching bandwidth than any previously demonstrated FPGA packet switch while consuming only one third the area.

In Part III, we developed a CAD system called `LYNX` to automatically connect FPGA applications using an embedded NoC. `LYNX` uses the connectivity graph of an application to decide how and where modules will connect to the NoC. The responsibilities of `LYNX` include: application clustering, module–router mapping, assigning the VC, configuring the FabricPort, generating any soft wrappers required, and generating the overall system HDL files for simulation and synthesis. Additionally, the NoC is itself parameterizable in `LYNX` through an extensible markup language (XML) file thus allowing NoC architecture exploration. The final topic we investigated in this thesis is transaction communication, and how to properly implement it using the embedded NoC. We automated the interconnection of transaction systems using `LYNX`, and compared our `LYNX` plus embedded NoC interconnection solution to Altera's Qsys and showed that we can achieve better latency and throughout in many cases.

In summary, our dissertation developed the different aspects of using embedded NoCs in FPGAs. We presented compelling solutions for the architecture, design and CAD of NoC-enhanced FPGAs, and we proved our propositions through application case studies and comparisons to existing interconnection solutions. Our results show that our NoCs can be a very useful embedded system-level interconnect to augment current FPGAs. Finally, we also created open-source tools and methodologies for the simulation, prototyping and CAD of embedded NoCs to encourage further research in this area.

14.2 Future Work

In this section, we list the avenues for research – building on the work presented in this thesis – that we believe are most promising.

14.2.1 LYNX Enhancements

In the immediate future, there are many enhancements that could be added to the LYNX CAD flow to make it more complete.

Multicast support

LYNX currently does not support multicast (sending the same data from one location to many). However, related work has shown that multicast is important for some FPGA applications [125, 126]. There are two basic methods by which embedded NoCs can support multicast. The first (and simpler) way is to send the same data multiple times, but this incurs time or resource penalties. This can be done by using a single NoC input, and sending the data serially (one-after-the-other). Alternatively, multiple NoC inputs can be used and the multicast data can be sent in parallel at the same time. The second multicast implementation method is to leverage previous research that investigated adding multicast support to NoC routers [60]. By adding multicast support to the routers themselves, we can avoid the time or resource penalty that can arise from the first method.

Detailed Application Specification

LYNX currently optimizes latency and throughput globally so that the overall performance of an application is maximized. However, this might ignore some detailed information about the application modules. For instance, if a module only sends data every 2 cycles (its initiation interval is equal 2), this information can be used to better optimize the application. Specifically, in this case, this would relax the throughput constraints on some NoC links during the mapping step for example. Other detailed information includes latency requirements (or the lack thereof) on certain connections or the entire application. This would also govern how modules are mapped to NoC routers and may better guide the LYNX flow. Additionally, LYNX will be able to output exactly whether each application constraint was achieved, or how close the tool came to fulfilling the requirements.

Benchmarks and Testing Infrastructure

We need to add more application benchmarks to LYNX. First, this is to track the performance and efficiency of LYNX' interconnect solutions for these applications. Second, like any CAD software, LYNX needs to undergo comprehensive testing every time its code is modified – the larger the benchmark test set, the better the quality of these tests. Unit testing the classes and functions of LYNX is also imperative as these tests are currently lacking. Ideally, an integrated build/test system should be used so that whenever new code is added, all LYNX unit and benchmarks tests would be run automatically.

NoC Architecture and I/Os

LYNX uses an NoC architecture that is entered through an XML file. This makes the used NoC very configurable; however, not all NoC parameters are currently exposed to the user. For example, the

routing function, NoC frequency and topology (we currently only support a mesh topology.) Additionally, we have no way to specify the I/O interfaces to which the NoC connects. The ability to better model an NoC architecture in LYNX will allow more detailed results and will also allow us to better explore the performance/efficiency of our benchmarks on different NoC architectures.

14.2.2 Mimic Benchmark Set

We described LYNXML and the Mimic flows in Chapter 12. The main idea is that we only need some application meta-data to generate and evaluate a system-level interconnect. Using an annotated ACG, we were able to generate NoC interconnection solutions from LYNX, and bus interconnects from Qsys and we compared the two in Chapter 13. We hope to extend that idea and create a comprehensive set of Mimic benchmarks that target different aspects of system-level interconnection. For example, one group of benchmarks can target streaming applications, while another group of benchmarks can target transaction-based communication, and a third benchmark group can combine the two ideas. Some more detailed benchmarks can also target interconnect capabilities such as correct ordering of transfers or multicast capability. By running the resulting benchmark suite through a system-level interconnect CAD tool, one can easily determine the capabilities of that tool, and more importantly, one can compare that tool easily against others.

We believe that this is the appropriate time to create such a comprehensive benchmark set for system-level interconnects. As FPGAs increase in size and complexity, interconnecting a complete application always entails some form of system-level interconnect. This is especially true for connecting fast I/Os and on-chip hard processors to applications configured using the FPGA fabric. Additionally, the research into system-level interconnection tools is on the rise. Among recent work is CONNECT [117], GENIE [125, 126], Hoplite [83] and our own tool LYNX. Each of those research tools rightfully claim their own merits – how do we compare them fairly to each other, and how do we compare them to industrial tools from Altera and Xilinx? We believe that a Mimic benchmark set is a good way to quantify the performance/efficiency merits of each CAD tool, and to easily compare the tools against each other.

14.2.3 Application Case Studies

In addition to the Mimic benchmarks, complete application case studies are required to more-accurately evaluate embedded NoCs. This also provides the opportunity to showcase the importance of embedded NoCs for certain classes of applications like we did for packet switching on FPGAs (Chapter 11) – in that case we showed that we can support more than 15× more bandwidth per area than previously demonstrated.

Hardware MapReduce

In the immediate future, candidate applications include hardware MapReduce [134]. This application requires a lot of arbitrary data movement between "map" and "reduce" kernels, since it is designed with processor clusters in mind. Therefore it assumes full connectivity between "map" and "reduce" kernels – this would map well to an embedded NoC-based crossbar. To scale a MapReduce application, one can typically just increase the number of available "map" and "reduce" kernels, perhaps beyond the size of a single FPGA device. This would be a good opportunity to explore how the NoC abstraction may

ease the design and interconnection of a multiple-FPGA application, and how that affects application performance.

Monte Carlo Simulation

Another important memory-intensive application is Monte Carlo simulation where large amounts of data is constantly being moved between on-chip compute kernels and to external memory [38]. In such an application transaction ordering is not always required for correctness and so it wold be interesting to evaluate a complete application without that constraint. Additionally, soft NoCs were previously tailored to implement Monte Carlo simulations efficiently [87] – it is important to compare our embedded NoC solution with their tailored soft NoC. This will allows us to determine whether we have indeed created enough flexibility in our embedded NoC to suit different applications, and will allow us to compare the efficiency and performance of our full-featured embedded NoC against a tailored soft NoC.

14.2.4 Virtualization with Embedded NoCs

Virtualization of FPGAs is an increasingly important research topic as FPGAs are being adopted in data centers [122]. We believe that NoCs have some attractive properties that can ease the virtualization of FPGAs. For example, we can use an embedded NoC to manage all communication between FPGA logic and I/O interfaces such as DDRx, Ethernet and PCIe. This will make it easier to create this application "shell" that is necessary for any application. Importantly, transfers between applications running on the FPGA and its I/Os can be managed securely by the communication abstraction provided by the NoC. For example, in the case of two different applications running on a virtualized FPGA, an NoC can use a firewall to guarantee that no application ever accesses data that belongs to a different application [93]. An embedded NoC can also encrypt all data transferred across its routers and links for extra security. Partial reconfiguration of applications onto virtualized FPGAs is another area that could be eased by the integration of an embedded NoC. We believe that the intersection of FPGA virtualization and embedded NoCs is a large and promising research space, and one that will grow in importance as FPGA use in data centers increases.

14.2.5 High-Level Synthesis with Embedded NoCs

HLS tools such as Leg-Up [63] generate hardware from a programming language such as C/C++. In doing so, the program functions become hardware "kernels" which are connected together using a form of system-level interconnect. We hope to leverage `LYNX` to interconnect such HLS systems together and compare overall system performance and efficiency with the current HLS interconnection solutions. This could be done by extending an open-source HLS tool such as Leg-Up to have the option to use `LYNX` as its interconnect tool; similar work has previously used soft NoCs to interconnect CUDA-generated HLS systems [43].

14.2.6 Partial Reconfiguration and Parallel Compilation

One of the main advantages of using an embedded NoCs is the modularity that it brings to hardware design on FPGAs – modules that are only connected through the NoC are decoupled as there are no timing paths between them. We believe this provides opportunities to make parallel compilation of these

modules much easier. Furthermore, because these modules are disjoint, it would be easier to swap them with other modules using partial reconfiguration. These two areas require further research to quantify the merits of using an NoC in their contexts.

14.2.7 Latency-Insensitive Design

In addition to the switching and arbitration capabilities of embedded NoCs, the can be used to simply improve timing on critical connections in design. In the traditional latency-insensitive design methodology [36], the timing-critical connections are mitigated by adding additional pipeline registers on them. Instead of re-pipelining a critical connection, we can map it onto an embedded NoC link.

14.2.8 Multi-Chip Interconnect and Interposers

Previous work has looked into easing the interconnection of multiple FPGAs by abstracting inter-FPGA communication using FIFOs [62]. We can extend that abstraction using an embedded NoC. Future research can create the design tools (or extend current ones such as `LYNX`) to partition a design over multiple FPGAs and connect through an embedded NoC that connects through FPGA I/Os. Silicon interposers are also of high interest in this context, as FPGAs are among the first semiconductor chips to leverage interposers to connect multiple FPGA dice in the same package [146]. This finer-grained multiple-FPGA interconnection could also be abstracted using an embedded NoC. Through the NoC abstraction, the only difference between intra- and inter-FPGA connections would be an increase in latency for connections between FPGAs dice. In both the multi-FPGA context, and the silicon interposer context, it would be interesting to compare an NoC-enhanced solution to current systems.

Appendix A

LYNXML Syntax

LYNXML is an XML description of an ACG. We use LYNXML to specify the connectivity between in a design, and we use that specification as input to our system-level interconnection tool: LYNX. We also hope that this syntax will be adopted by other system-level interconnection tools to be able to compare easily across different tools. We used XML to keep the syntax simple and easily extensible, additionally, there are many XML parsers available making it very easy to add LYNXML support to existing or new CAD tools.

This appendix documents the current syntax of LYNXML. The main constructs in LYNXML are <module>, <bundle> and <connection> – as their names suggest, these XML tags describe design modules, their bundles and connections between bundles. In the following, we describe the XML tags and attributes that define the LYNXML specification.

Tag	<design>
Nested Tags	<module>, <connection>
Attributes	[name]
Example	<design name="mydes">...</design>
Description	A design encapsulates all modules and connections that describe an application. It can optionally have an arbitrary string as its name. The name is the top-level design name in HDL generation.

Tag	<module>
Nested Tags	<bundle>, <port>, <parameter>
Attributes	type, name
Example	<module type="myadd" name="myadd_a">...</module>
Description	A module is a logical entity that contains part of a design. The module has two mandatory attributes. Type denotes the module's hardware block type; for example, the module name in Verilog. Note that the type should match the Verilog/VHDL module name if an implementation for it exists. The name attribute must be unique in the scope of a design and can be any arbitrary string that uniquely identifies each module instance.

Tag	<port>
Nested Tags	--
Attributes	name, [width], direction, type, [global]
Example	<port name="idata_in" width="128" direction="input" type="data"/> <port name="rst" direction="input" type="rst" global="rst"/>
Description	Ports correspond to input and output ports that exists in HDLs such as Verilog/VHDL. It is the basic communication primitive between modules. In a module's scope, the name attribute is a unique string that denotes a port. Width is the port width; if omitted, a default width of 1 is used. Direction specifies whether this is an input or output port. Type is the port type; we currently have 6 different types: clk, rst, ready, valid, data and dst (destination). The global attribute is used when a signal should be exported to the top level design to become a chip-level I/O – the second example shows how this attribute can be used with the reset signal.

Tag	<bundle>
Nested Tags	<port>
Attributes	name
Example	<bundle name="obun0">...</bundle>
Description	Bundles group data, ready and valid ports together. Any bundle must contain at least one data port, and exactly one ready signal and one valid signal. A bundle is a latency-insensitive communication endpoint between two modules. It must have a name attribute that is unique in the scope of the module to which it belongs.

Tag	<connection>
Nested Tags	--
Attributes	start, end
Example	<connection start="myadd_a.obun0" end="mymul_b.ibun0"/>
Description	A connection tag specifies a pair of bundles that are connected to each other. Note that any connection must start at an output bundle and must end at an input bundle.

Bibliography

[1] *International technology Roadmap for Semiconductors (ITRS)*, 2013.

[2] Mohamed S Abdelfattah and Vaughn Betz. Design Tradeoffs for Hard and Soft FPGA-based Networks-on-Chip. In *International Conference on Field-Programmable Technology (FPT)*, pages 95–103. IEEE, 2012.

[3] Mohamed S. Abdelfattah and Vaughn Betz. Augmenting FPGAs with Embedded Networks-on-Chip. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, 2013.

[4] Mohamed S Abdelfattah and Vaughn Betz. The Power of Communication: Energy-Efficient NoCs for FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013.

[5] Mohamed S Abdelfattah and Vaughn Betz. Networks-on-Chip for FPGAs: Hard, Soft or Mixed? *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–22, 2014.

[6] Mohamed S Abdelfattah and Vaughn Betz. The Case for Embedded Networks on Chip on FPGAs. *IEEE Micro*, 34(1):80–89, 2014.

[7] Mohamed S Abdelfattah and Vaughn Betz. Field Programmable Gate-Array with Network-on-Chip Hardware and Design Flow, 04 2015. US Patent Application 14/060,253.

[8] Mohamed S Abdelfattah and Vaughn Betz. Embedded Networks-on-Chip for FPGAs. In Pierre-Emmanuel Gaillardon, editor, *Reconfigurable Logic: Architecture, Tools and Applications*, chapter 6, pages 149–184. CRC Press, 2016.

[9] Mohamed S Abdelfattah and Vaughn Betz. LYNX: CAD for Embedded NoCs on FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2016.

[10] Mohamed S Abdelfattah and Vaughn Betz. Power Analysis of Embedded NoCs on FPGAs and Comparison With Custom Buses. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 24(1):165–177, 2016.

[11] Mohamed S. Abdelfattah, Andrew Bitar, and Vaughn Betz. Take the Highway: Design for Embedded NoCs on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 98–107. ACM, 2015.

[12] Mohamed S Abdelfattah, Andrew Bitar, and Vaughn Betz. Design and Applications for Embedded Networks-on-Chip on Field-Programmable Gate-Arrays. *IEEE Transactions on Computers (TCOMP)*, 2016.

[13] M.S. Abdelfattah, A. Bitar, A. Yaghi, and V. Betz. Design and simulation tools for Embedded NOCs on FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2015. [Demonstration Abstract].

[14] Altera Corp. Stratix PowerPlay Early Power Estimator.

[15] Altera Corp. Stratix III FPGA: Lowest Power, Highest Performance 65-nm FPGA. Press Release, 2007.

[16] Altera Corp. High-Definition Video Reference Design (UDX6), 2013.

[17] Altera Corp. Video and Image Processing Suite User Guide, 2014.

[18] Altera Corp. *External Memory Interface Handbook*, November 2015.

[19] Altera Corp. *UGPARTRECON - Partial Reconfiguration IP Core*, May 2015.

[20] M. An, J. G. Steffan, and V. Betz. Speeding Up FPGA Placement: Parallel Algorithms and Methods. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 178–185. IEEE, 2014.

[21] Andrew Bitar. Building Networking Applications from a NoC-Enhanced FPGA. Master's thesis, University of Toronto, 2015.

[22] F. Angiolini, P. Meloni, S.M. Carta, L. Raffo, and L. Benini. A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MPSoCs. volume 26, pages 421–434, 2007.

[23] Federico Angiolini, Paolo Meloni, Salvatore Carta, Luca Benini, and Luigi Raffo. Contrasting a NoC and a traditional interconnect fabric with layout awareness. In *Design Automation and Test in Europe (DATE)*, pages 124–129. ACM, 2006.

[24] Gregg Baeckler. Conference Keynote: HyperPipelining of High-Speed Interface Logic. International Symposium on Field-Programmable Gate Arrays (FPGA), 2016.

[25] James Balfour and William J. Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 187–198. ACM, 2006.

[26] T A Bartic, J Mignolet, V Nollet, T Marescaux, D Verkest, S Vernalde, and R Lauwereins. Topology adaptive network-on-chip design and implementation. *IEEE Computers and Digital Techniques*, 152(4):467–472, 2005.

[27] Daniel U Becker and William J Dally. Allocator Implementations for Network-on-Chip Routers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. ACM/IEEE, 2009.

[28] Luca Benini. Application specific NoC design. pages 105–111. ACM, 2006.

[29] Luca Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[30] Andrew Bitar, Mohamed S. Abdelfattah, and Vaughn Betz. Bringing Programmability to the Data Plane: Packet Processing with a NoC-Enhanced FPGA. In *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2015.

[31] Andrew Bitar, Jeffrey Cassidy, Natalie Enright Jerger, and Vaughn Betz. Efficient and programmable Ethernet switching with a NoC-enhanced FPGA. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, 2014.

[32] M. T. Bohr. Interconnect scaling-the real limiter to high performance ULSI. In *International Electron Devices Meeting (IEDM)*, pages 241–244. IEEE, 1995.

[33] Misha Burich. Conference Workshop: FPGAs in 2032, Challenges and Opportunities in the next 20, years – Convergence of Programmable Solutions. International Symposium on Field-Programmable Gate Arrays (FPGA), 2012.

[34] L. P. Carloni. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proceedings of the IEEE*, 103(11):2133–2151, 2015.

[35] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *International Conference on Computer Aided Design (ICCAD)*, pages 309–315. IEEE, 1999.

[36] Luca Carloni and Alberto Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, 2002.

[37] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance Analysis and Optimization of Latency Insensitive Systems. In *Design Automation Conference (DAC)*, pages 361–367. IEEE, 2000.

[38] J. Cassidy, L. Lilge, and V. Betz. Fast, Power-Efficient Biophotonic Simulations for Cancer Treatment Using FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140. ACM, May 2014.

[39] Cavium. XPliant Ethernet Switch Product Family, 2014.

[40] Shant Chandrakar, Dinesh Gaitonde, and Trevor Bauer. Enhancements in UltraScale CLB Architecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 108–116. ACM, 2015.

[41] D. Chen and D. Singh. Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 5–12. IEEE, 2012.

[42] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platform. In *Design Automation Conference (DAC)*, pages 297–304. IEEE, 2013.

[43] Y. Chen, S. T. Gurumani, Y. Liang, G. Li, D. Guo, K. Rupnow, and D. Chen. FCUDA-NoC: A Scalable and Efficient Network-on-Chip Implementation for the CUDA-to-FPGA Flow. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, PP(99):1–14, 2015.

[44] C. Chiasson and V. Betz. Should FPGAS abandon the pass-gate? In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013.

[45] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip H.W. Leong, and David B. Thomas. A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 57–66. ACM, 2012.

[46] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 97–106. ACM, 2011.

[47] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and Evaluation of the CoRAM Memory Architecture for FPGA-based Computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 139–142. ACM, 2012.

[48] J. Cong. An interconnect-centric design flow for nanometer technologies. *Proceedings of the IEEE*, 89(4):505–528, 2001.

[49] Altera Corp. Altera Qsys.

[50] Zefu Dai and Jianwen Zhu. Saturating the Transceiver BW: Switch Fabric Design on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 67–75. ACM, 2012.

[51] W. J. Dally and S. Lacy. VLSI architecture: past, present, and future. In *20th Anniversary Conference on Advanced Research in VLSI*, pages 232–241. IEEE, 1999.

[52] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, Boston, MA, 2004.

[53] W.J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Design Automation Conference (DAC)*, pages 684–689. IEEE, 2001.

[54] Daniel U. Becker. *Efficient Microarchitecture for Network-on-Chip Router*. PhD thesis, Stanford University, 2012.

[55] T. Dorta, J. Jimnez, J. L. Martn, U. Bidarte, and A. Astarloa. Overview of FPGA-Based Multiprocessor Systems. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 273–278. IEEE, 2009.

[56] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 126–135. Springer-Verlag, 1996.

[57] Andreas Ehliar and Dake Liu. An FPGA Based Open Source Network-on-Chip Architecture. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 800–803. IEEE, 2007.

[58] I. Elhanany, D. Chiou, V. Tabatabaee, R. Noro, and A. Poursepanj. The network processing forum switch fabric benchmark specifications: An overview. *IEEE Network*, 19(2):5–9, 2005.

[59] Natalie Enright Jerger and Li-Shiuan Peh. *On-Chip Networks*. Morgan Claypool, 2009.

[60] Natalie Enright Jerger, Li-Shiuan Peh, and Mikko Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *International Symposium on Computer Architecture (ISCA)*, pages 229–240. ACM, 2008.

[61] K. Fleming, H. J. Yang, M. Adler, and J. Emer. The LEAP FPGA operating system. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[62] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging Latency-insensitivity to Ease Multiple FPGA Design. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 175–184. ACM, 2012.

[63] B. Fort, A. Canis, J. Choi, N. Calagar, Ruolong Lian, S. Hadjis, Yu Ting Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. In *International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 120–129, 2014.

[64] R. Francis and S. Moore. Exploring hard and soft networks-on-chip for FPGAs. In *International Conference on Field-Programmable Technology (FPT)*, pages 261–264. IEEE, 2008.

[65] R. Francis, S. Moore, and R. Mullins. A Network of Time-Division Multiplexed Wiring for FPGAs. In *International Symposium on Networks-on-Chip (NOCS)*, pages 35–44. ACM/IEEE, 2008.

[66] R. Gindin, I. Cidon, and I. Keidar. NoC-Based FPGA: Architecture and Routing. In *International Symposium on Networks-on-Chip (NOCS)*, pages 253–264. ACM/IEEE, 2007.

[67] K. Goossens, A. Radulescu, and A. Hansson. A unified approach to constrained mapping and routing on network-on-chip architectures. In *International Conference on Hardware/Software Codesign and System Synthesis, 2005. (CODES+ISSS)*, pages 75–80. IEEE/ACM/IFIP, 2005.

[68] Kees Goossens, John Dielissen, and Andrei Radulescu. AEThereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test*, 22(5), 2005.

[69] Kees Goossens et al. Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In *International Symposium on Networks-on-Chip (NOCS)*, pages 45–54. ACM/IEEE, 2008.

[70] G. Guindani, C. Reinbrecht, T. Raupp, N. Calazans, and F.G. Moraes. NoC Power Estimation at the RTL Abstraction Level. In *Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 475–478. IEEE, 2008.

[71] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann. Dynamic reconfiguration with hardwired networks-on-chip on future FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 527–530. IEEE, 2005.

[72] Ahmed Hemani, Axel Jantsch, Shashi Kumar, Adam Postula, Jonny Oberg, Mikael Millberg, and Dan Lindqvist. Network on a Chip: An architecture for billion transistor era . In *IEEE Norchip Conference*, pages 1–8. IEEE, 2000.

[73] Andy Henson and Richard Herveille. Video Compression Systems. `www.opencores.org/project, video_systems`, 2008.

[74] Clint Hilton and Brent Nelson. A Flexible Circuit-Switched NoC For FPGA-based Systems. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 191–196. IEEE, 2006.

[75] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[76] Jincao Hu and Radu Marculescu. Application-specific buffer space allocation for networks-on-chip router design. In *International Conference on Computer Aided Design (ICCAD)*, pages 354–361. IEEE, 2004.

[77] Jingcao Hu and Radu Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. *Design Automation and Test in Europe (DATE)*, pages 688–693, 2003.

[78] Y. Huan and A. DeHon. FPGA Optimized Packet-Switched NoC using Split and Merge Primitives. In *International Conference on Field-Programmable Technology (FPT)*, pages 47–52. IEEE, 2012.

[79] Xilinx Inc. Vivado IP Integrator.

[80] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96. IEEE, 2013.

[81] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on FPGAs. In *FPGA*, pages 219–228, 2009.

[82] Andrew B Kahng, Bin Li, Li-shiuan Peh, and Kambiz Samadi. ORION 2.0 : A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Design Automation and Test in Europe (DATE)*, pages 0–5. ACM, 2009.

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

[84] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed fpga overlay networks. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 205–216. ACM, 2006.

[85] Kent J. Orthner. Packet-Based Transaction Interconnect Fabric for FPGA Systems on Chip. Master's thesis, Carleton University, 2009.

[86] J. Kim. Low-cost router microarchitecture for on-chip networks. In *International Symposium on Microarchitecture*, pages 255–266. IEEE/ACM, 2009.

[87] P. J. Kinsman and N. Nicolici. NoC-Based FPGA Acceleration for Monte Carlo Simulations with Applications to SPECT Imaging. *IEEE Transactions on Computers*, 62(3):524–535, March 2013.

[88] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Computer Society Annual Symposium on VLSI*, pages 105–112. IEEE, 2002.

[89] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits (TCAD)*, 26(2):203–215, 2007.

[90] A. Lambrechts, P. Raghavan, A. Leroy, G. Talavera, T.V. Aa, M. Jayapala, F. Catthoor, D. Verkest, G. Deconinck, H. Corporaal, F. Robert, and J. Carrabina. Power breakdown analysis for a heterogeneous NoC running a video application. In *International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 179–184. IEEE, 2005.

[91] Martin Langhammer and Bogdan Pasca. Floating-Point DSP Block Architecture for FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 117–125. ACM, 2015.

[92] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using Hard Macros to Reduce FPGA Compilation Time. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 438–441. IEEE, 2010.

[93] Jean-Jacques Lecler and Gilles Baillieu. Application driven network-on-chip architecture exploration and refinement for a complex SoC. *Design and Automation of Embedded Systems*, 15(2):133–158, 2011.

[94] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras, and Radu Marculescu. On-chip Communication Architecture Exploration: A Quantitative Evaluation of Point-to-point, Bus, and Network-on-chip Approaches. *ACM Transactions on Design Automation Electronic Systems (TODAES)*, 12(3):23:1–23:20, 2008.

[95] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The Stratix II Logic and Routing Architecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 14–20. ACM, 2005.

[96] David Lewis, Elias Ahmed, David Cashman, Tim Vanderhoek, Chris Lane, Andy Lee, and Philip Pan. Architectural Enhancements in Stratix-III™and Stratix-IV™. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 33–42. ACM, 2009.

[97] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. The Stratix Routing and Logic Architecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 12–20. ACM, 2003.

[98] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural Enhancements in Stratix V. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 147–156. ACM, 2013.

[99] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The Stratix™10 Highly Pipelined FPGA Architecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 159–168. ACM, 2016.

[100] Zhuo Li and Charles J. Alper. What is Physical Synthesis? *ACM/SIGDA E-Newsletter*, 41(1), 2011.

[101] O. Lindtjorn, R. Clapp, O. Pell, Haohuan Fu, M. Flynn, and Haohuan Fu. Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. *IEEE Micro*, 31(2):41–49, 2011.

[102] Ruibing Lu and Cheng-Kok Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *International Conference on Computer Aided Design (ICCAD)*, pages 227–231. IEEE, 2003.

[103] Ruibing Lu and Cheng-Kok Koh. Performance analysis of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits (TCAD)*, 25(3):469–483, 2006.

[104] Ting Lu, Ryan Kenny, and Sean Atsatt. Stratix 10 Secure Device Manager Provides Best-in-Class FPGA and SoC Security. Technical report, Altera Corp., 06 2015.

[105] Z. Lu, L. Xia, and A. Jantsch. Cluster-based Simulated Annealing for Mapping Cores onto 2D Mesh Networks on Chip. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 1–6. IEEE, April 2008.

[106] Adrian Ludwin and Vaughn Betz. Efficient and Deterministic Parallel Placement for FPGAs,. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(3):1–23, 2011.

[107] Terrence S. T. Mak, Pete Sedcole, Peter Y. K. Cheung, and Wayne Luk. On-FPGA Communication Architectures and Design Factors. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2006.

[108] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 595–605. Springer, 2003.

[109] Theodore Marescaux, Andrei Bartic, Diderick Verkest, D. Verkest, Rudy Lauwereins, Serge Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking On FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 795–805, 2002.

[110] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[111] R. Mullins. Minimising Dynamic Power Consumption in On-Chip Networks. In *International Symposium on System-on-Chip (ISSoC)*, pages 1–4. IEEE, 2006.

[112] Robert Mullins, Andrew West, and Simon Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *International Symposium on Computer Architecture (ISCA)*, pages 188–198. ACM, 2004.

[113] S Murali, Luca Benini, and Giovanni De Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 27–32. IEEE, 2005.

[114] S. Murali and G. De Micheli. SUNMAP: a tool for automatic topology selection and generation for NoCs. In *Design Automation Conference (DAC)*, pages 914–919. ACM, 2004.

[115] Kevin E. Murray and Vaughn Betz. Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 223–232. ACM, 2014.

[116] P.P. Pande, C. Grecu, a. Ivanov, R. Saleh, and G. De Micheli. Design, Synthesis, and Test of Networks on Chips. *IEEE Design and Test of Computers*, 22(5):404–413, May 2005.

[117] Michael K Papamichael and James C Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 37–46. ACM, 2012.

[118] Giorgos Passas, Manolis Katevenis, and Dionisios Pnevmatikatos. Crossbar NoCs Are Scalable Beyond 100 Nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits (TCAD)*, 31(4):573–585, 2012.

[119] Li-Shiuan Peh and William J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 255–267. IEEE, 2001.

[120] Li-Shiuan Peh and Natalie Enright Jerger. *On-Chip Networks*. Morgan and Claypool Publishers, 2009.

[121] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft connections: Addressing the hardware-design modularity problem. In *Design Automation Conference (DAC)*, pages 276–281. IEEE, 2009.

[122] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*. ACM, June 2014.

[123] Jan Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits, A Design Perspective*. Pearson Education, Inc., Upper Saddle River, NJ, 2 edition, 2003.

[124] R. Rashid, J.G. Steffan, and V. Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *International Conference on Field-Programmable Technology (FPT)*, pages 20–27. IEEE, 2014.

[125] Alex Rodionov and Jonathan Rose. Automatic FPGA system and interconnect construction with multicast and customizable topology. In *International Conference on Field-Programmable Technology (FPT)*, pages 72–79. IEEE, 2015.

[126] Alex Rodionov and Jonathan Rose. Fine-Grained Interconnect Synthesis. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 46–55. ACM, 2015.

[127] Pradip Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1):60 – 76, 2013.

[128] Manuel Saldaña, Lesley Shannon, and Paul Chow. The Routability of Multiprocessor Network Topologies in FPGAs. In *International Workshop on System-level Interconnect Prediction (SLIP)*, pages 49–56. ACM, 2006.

[129] E. Salminen, A. Kulmala, and T. D. Hamalainen. HIBI-based multiprocessor SoC on FPGA. In *International Symposium on Circuits and Systems (ISCAS)*, pages 3351–3354. IEEE, 2005.

[130] E. Salminen, A. Kulmala, and T. D. Hamalainen. On network-on-chip comparison. In *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 503–510. IEEE, 2007.

[131] Graham Schelle and Dirk Grunwald. Exploring FPGA network on chip implementations across various application and network loads. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 41–46. IEEE, 2008.

[132] Ryan Scoville. TimeQuest User Guide, 2010.

[133] Balasubramanian Sethuraman, Prasun Bhattacharya, Jawad Khan, and Ranga Vemuri. LiPaR: A Light-Weight Parallel Router for FPGA-based Networks-on-Chip. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 452–457. ACM, 2005.

[134] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce framework on FPGA. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 93–102. ACM, 2010.

[135] Akbar Sharifi, Asit K. Mishra, Shekhar Srikantaiah, Mahmut Kandemir, and Chita R. Das. PE-PON: Performance-aware Hierarchical Power Budgeting for NoC Based Multicores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 65–74. ACM, 2012.

[136] Deshanand P. Singh and Stephen D. Brown. The Case for Registered Routing Switches in Field Programmable Gate Arrays. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–169. ACM, 2001.

[137] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[138] Synopsys Inc. *Design Compiler Optimization Reference Manual*. 2010.

[139] Yuval Tamir and Gregory L. Frazier. High-Performance Multi-Queue Buffers for VLSI Communication Switches. In *International Symposium on Computer Architecture (ISCA)*, pages 343–354. ACM, 1988.

[140] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[141] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[142] R. Thid, I. Sander, and A. Jantsch. Flexible Bus and NoC Performance Analysis with Configurable Synthetic Workloads. In *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 681–688. IEEE, 2006.

[143] L G Valiant and G J Brebner. Universal schemes for parallel communication. In *ACM Symposium on Theory of Computing (STOC)*, pages 263–277. ACM, 1981.

[144] Hang-Sheng Wang, Li-Shiuan Peh, and S. Malik. A power model for routers: modeling Alpha 21364 and InfiniBand routers. *IEEE Micro*, 23(1):26–35, 2003.

[145] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 5–14. ACM, 2011.

[146] Xilinx Inc. Virtex-5,6,7 Family Overview, 2009-2014.

[147] Xilinx Inc. *UG702 - Partial Reconfiguration User Guide*, April 2012.

[148] Xilinx Inc. *UltraScale Architecture FPGAs Memory IP*, November 2015.

[149] Xilinx Inc. *xtp414 - Ultrascale Maximum Memory Performance Utility*, September 2015.

[150] A. Ye and J. Rose. Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 14(5):462–473, 2006.

[151] T. T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Design Automation Conference (DAC)*, pages 524–529. IEEE, 2002.

[152] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin. A Study on Communication Issues for Systems-on-Chip. In *Symposium on Circuits and Systems Design (ICSD)*, pages 121–126. IEEE, 2002.

[153] Cesar Albenes Zeferino, Marcio Eduardo Kreutz, and Altamiro Amadeu Susin. RASoC: A Router Soft-Core for Networks-on-Chip. In *Design Automation and Test in Europe (DATE)*, volume 3, pages 198–203. ACM, 2004.