UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática
de Sistemas y Computadores

Conflict-Free Networks on Chip for Real Time Systems

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Computer Engineering)

Author

Tomás Picornell Sanjuan

Advisors

Prof. José Flich Cardo
Dr. Carles Hernández Luz

October 2021

Doctoral Committee

- Prof. Pedro J. López Rodríguez

  Universitat Politècnica de València, Valencia, Spain

- Prof. José Cano Reyes

  University of Glasgow, Glasgow, UK

- Prof. Giorgos Dimitrakopoulos

  University of Thrace, Xanthi, Greece

Agradecimientos

Hay personas que marcan tu vida, que te ayudan a mantener el rumbo o que te inspiran para cambiarlo. Es por eso que quiero expresar mi profundo agradecimiento a todas ellas. Todo empezó en segundo de carrera cuando un profesor de la asignatura despertó mi inquietud. Al poco tiempo de conocerle y ver los conocimientos y la temática en la que se desenvolvía, le dije que me gustaría acabar trabajando en una de las mayores empresas del sector de arquitectura de computadores. Él me respondió que ello era posible, y me ofreció la oportunidad de hacer el doctorado.

En primer lugar quiero empezar agradeciendo la especial confianza que ha tenido Pepe, el director de mi tesis, durante todos estos años. Por mantener mi rumbo, por todo su esfuerzo y tener la paciencia necesaria para enseñarme todo lo que sé de arquitectura de computadores y la investigación. Otra de esas personas a las que quiero mostrar mi especial agradecimiento es a Carles. Por darme ese empujón que me faltaba en el mundo de la investigación, ya que desde ahí todo fue mas fácil. También agradecer a José Duato por su colaboración en mi tesis, por sus ideas tan brillantes, así como por ayudarme a obtener la beca que me ha dado la posibilidad de realizar este doctorado. Es cierto que sin ellos esta tesis no sería posible.

En segundo lugar quiero agradecer a todos mis compañeros que he tenido durante el desarrollo de la tesis al grupo de arquitecturas paralelas por contribuir a un ambiente de trabajo tan bueno y por todos los momentos que hemos compartido juntos. En especial a Rafa y a Chema del PEAK Team. A Rafa, por estar siempre literalmente a mi lado y compartir todos sus conocimientos. A Chema también por sus conocimientos, pero en especial, por todos los cafés que hemos compartido así como comidas, cervezas e inquietudes.

Por último, pero no menos importante, a mi familia por lo que habéis hecho por mí. En especial a mi madre y a mi padre. A mi madre, por exigir siempre llegar un poco mas lejos en mis estudios. A mi padre, por apoyarme en todo lo que me he propuesto. A mi pareja, por apoyarme y aguantarme siempre en todos aquellos momentos difíciles. Ella es la persona que sabe cambiar mi estado de ánimo y hace que pueda salir adelante.

Y finalmente a mis amigos y compañeros por haberme ayudado a desconectar en esos momentos de deporte y ocio tan importantes para mí.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
BE	**B**est **E**ffort
CAGR	**C**ompound **A**nnual **G**rowth **R**ate
CS	**C**ircuit **S**witching
CDG	**C**hannel **D**ependency **G**raph
COTS	**C**omercial **O**ff-**T**he-**S**helf
DI	**D**omain **I**dentifier
DOR	**D**imension **O**rder **R**outing
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
FPA	**F**ixed **P**riority **A**rbiter
FPGA	**F**ield **P**rogramable **G**ate **A**rray
GAP	**G**rupo de **A**rquitecturas **P**aralelas
GS	**G**uaranteed **S**ervices
HDL	**H**ardware **D**escription **L**anguage
HOV	**H**ighest **O**bserved **V**alue
IC	**I**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
L1	First-level (cache)
L2	Second-level (cache)
LFSR	**L**inear **F**eedback **S**hift **R**egisters
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
PS	**P**acket **S**witching
PEAK	**P**artitioned-**E**nabled **A**rchitecture for **K**ilocores
QoS	**Q**uality of **S**ervice
MC	**M**emory **C**ontroller

MCD	**M**aximum **C**ommon **D**ivisor
MS	**M**essage **S**ystem Generator
NCA	**N**on-**C**acheable **A**ddresses
NI	**N**etwork **I**nterface
NIC	**N**etwork **I**nterface **C**ontroller
NoC	**N**etwork-**o**n-**C**hip
OSR	**O**verlapped **S**tatic **R**econfiguration
QoS	**Q**uality **o**f **S**ervice
RTL	**R**egister-**T**ransfer **L**evel
RTT	**R**ound-**T**rip **T**ime
RST	**R**oute **S**cheduling **T**able
SA	**S**witch **A**locator
SAF	**S**tore **A**nd **F**orward
SoC	**S**ystem-**o**n-**C**hip
SR	**S**egment-based **R**outing
TCL	**T**ool **C**ommand **L**anguage
TDM	**T**ime **D**ivision **M**ultiplexing
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
VC	**V**irtual **C**hannel
VCT	**V**irtual **C**ut-**T**hrough
VN	**V**irtual **N**etwork
WCET	**W**orst **C**ase **E**xecution **T**ime
WH	**W**orm**h**ole

Abstract

The ever need for higher performance to cope with the high computational power demands of new applications (e.g autonomous driving systems), forces industry to support technology based on multi-processors system on chip (MPSoCs) in their safety-critical embedded systems. MPSoCs usually include a network-on-chip (NoC) to interconnect the cores between them and, with memory and the rest of shared resources. Unfortunately, the inclusion of NoCs difficults achieving time predictability as network-level conflicts may occur in many points in a distributed manner.

To overcome this problem, this thesis proposes a new time-predictable NoC design paradigm where conflicts within the network are eliminated by design. This new paradigm builds on top of the Channel Dependency Graph (CDG) in order to deterministically avoid network conflicts. Our solution is able to naturally inject messages using a TDM period equal to the optimal theoretical bound without the need of using a computationally demanding offline process. The network is integrated in a tile-based manycore system and adapted to its memory hierarchy.

As a second main contribution, we propose a novel distributed dynamic scheduler that is able to achieve peak performance close to a wormhole-based NoC design without compromising its real-time guarantees. The scheduler builds on top of our NoC design to exploit its key properties.

The results of our NoC show that our design guarantees time predictability avoiding network interference among multiple running applications. The network always guarantees performance and also improves wormhole performance in a 4×4 setting by a factor of $3.7\times$ when interference traffic is injected. For a 8×8 network differences are even larger. In addition, the network obtains a total area saving of 10.79% over a standard wormhole implementation.

The proposed scheduler achieves an overall throughput improvement of $6.9\times$ and $14.4\times$ over a baseline conflict-free NoC for 16 and 64-node meshes, respectively. When compared against a standard wormhole router 95% of its network throughput is preserved

while strict timing predictability is kept. This achievement opens the door to new high performance time predictable NoC designs.

As a final contribution, we build a taxonomy of TDM-based NoCs with real-time properties. With this taxonomy we perform a comprehensive analysis to study and compare from response time specific, to low resource implementation cost, through trade-off solutions for real-time NoCs designs. As a result, we derive new TDM-based NoC designs.

Resumen

La constante necesidad de un mayor rendimiento para cumplir con la gran demanda de potencia de cómputo de las nuevas aplicaciones, (ej. sistemas de conducción autónoma), obliga a la industria a apostar por la tecnología basada en Sistemas en Chip con Procesadores Multinúcleo (MPSoCs) en sus sistemas embebidos de seguridad-crítica. Los sistemas MPSoCs generalmente incluyen una red en el chip (NoC) para interconectar los núcleos de procesamiento entre ellos, con la memoria y con el resto de recursos compartidos. Desafortunadamente, el uso de las NoCs dificulta alcanzar la predecibilidad en el tiempo, ya que pueden aparecer conflictos en muchos puntos y de forma distribuida a nivel de red.

Para afrontar este problema, en esta tesis se propone un nuevo paradigma de diseño para NoCs de tiempo real donde los conflictos en la red son eliminados por diseño. Este nuevo paradigma parte del Grafo de Dependencia de Canales (CDG) para evitar los conflictos de red de forma determinista. Nuestra solución es capaz de inyectar mensajes de forma natural usando un periodo TDM igual al límite teórico óptimo sin la necesidad de usar un proceso offline exigente computacionalmente. La red se ha integrado en un sistema multinúcleo basado en *tiles* y adaptado a su jerarquía de memoria.

Como segunda contribución principal, proponemos un nuevo planificador dinámico y distribuido capaz de alcanzar un rendimiento pico muy cercanos a las NoC basadas en un diseño *wormhole* sin comprometer sus garantías de tiempo real. El planificador se basa en nuestro diseño de red para explotar sus propiedades clave.

Los resultados de nuestra NoC muestran que nuestro diseño garantiza la predecibilidad en el tiempo evitando interferencias en la red entre múltiples aplicaciones ejecutándose concurrentemente. La red siempre garantiza el rendimiento y también mejora el rendimiento respecto al de las redes *wormhole* en una red 4×4 en un factor de $3,7\times$ cuando se inyecta trafico para generar interferencias. En una red 8×8 las diferencias son incluso mayores. Además, la red obtiene un ahorro de área total del $10,79\,\%$ frente a una implementación básica de una red *wormhole*.

El planificador propuesto alcanza una mejora de rendimiento de $6,9\times$ y $14,4\times$ frente la versión básica de la red DCFNoC para redes en forma de malla de 16 y 64 nodos, respectivamente. Cuando lo comparamos frente a un conmutador estándar *wormhole* se preserva un rendimiento de red del 95 % al mismo tiempo que preserva la estricta predecibilidad en el tiempo. Este logro abre la puerta a nuevos diseños de NoCs de alto rendimiento con predecibilidad en el tiempo.

Como contribución final, construimos una taxonomía de NoCs basadas en TDM con propiedades de tiempo real. Con esta taxonomía realizamos un análisis exhaustivo para estudiar y comparar desde tiempos de respuesta, a implementaciones con bajo coste, pasando por soluciones de compromiso para diseños de NoCs de tiempo real. Como resultado, obtenemos nuevos diseños de NoCs basadas en TDM.

Resum

La constant necessitat d'un major rendiment per a complir amb la gran demanda de potència de còmput de les noves aplicacions, (ex. sistemes de conducció autònoma), obliga la indústria a apostar per la tecnologia basada en Sistemes en Xip amb Processadors Multinucli (MPSoCs) en els seus sistemes embeguts de seguretat-crítica. Els sistemes MPSoCs generalment inclouen una xarxa en el xip (NoC) per a interconnectar els nuclis de processament entre ells, amb la memòria i amb la resta de recursos compartits. Desafortunadament, l'ús de les NoCs dificulta aconseguir la predictibilitat en el temps, ja que poden aparéixer conflictes en molts punts i de forma distribuïda a nivell de xarxa.

Per a afrontar aquest problema, en aquesta tesi es proposa un nou paradigma de disseny per a NoCs de temps real on els conflictes en la xarxa són eliminats per disseny. Aquest nou paradigma parteix del Graf de Dependència de Canals (CDG) per a evitar els conflictes de xarxa de manera determinista. La nostra solució és capaç d'injectar missatges de mra natural fent ús d'un període TDM igual al límit teòric òptim sense la necessitat de fer ús d'un procés offline exigent computacionalment. La xarxa s'ha integrat en un sistema multinucli basat en *tiles* i adaptat a la seua jerarquia de memòria.

Com a segona contribució principal, proposem un nou planificador dinàmic i distribuït capaç d'aconseguir un rendiment pic molt pròxims a les NoC basades en un disseny *wormhole* sense comprometre les seues garanties de temps real. El planificador es basa en el nostre disseny de xarxa per a explotar les seues propietats clau.

Els resultats de la nostra NoC mostren que el nostre disseny garanteix la predictibilitat en el temps evitant interferències en la xarxa entre múltiples aplicacions executant-se concurrentment. La xarxa sempre garanteix el rendiment i també millora el rendiment respecte al de les xarxes *wormhole* en una xarxa 4×4 en un factor de $3,7\times$ quan s'injecta trafic per a generar interferències. En una xarxa 8×8 les diferències són fins i tot majors. A més, la xarxa obté un estalvi d'àrea total del $10,79\%$ front una implementació bàsica d'una xarxa *wormhole*.

El planificador proposat aconsegueix una millora de rendiment de $6,9\times$ i $14,4\times$ front la versió bàsica de la xarxa DCFNoC per a xarxes en forma de malla de 16 i 64 nodes, respectivament. Quan ho comparem amb un commutador estàndard *wormhole* es preserva un rendiment de xarxa del 95% al mateix temps que preserva la estricta predictibilitat en el temps. Aquest assoliment obri la porta a nous dissenys de NoCs d'alt rendiment amb predictibilitat en el temps.

Com a contribució final, construïm una taxonomia de NoCs basades en TDM amb propietats de temps real. Amb aquesta taxonomia realitzem una anàlisi exhaustiu per a estudiar i comparar des de temps de resposta, a implementacions amb baix cost, passant per solucions de compromís per a dissenys de NoCs de temps real. Com a resultat, obtenim nous dissenys de NoCs basades en TDM.

Chapter 1

Introduction

The market of embedded systems has been showing a constant growth over the last years. According to the report by Transparency Market Research [4], embedded systems market revenue will rise to US$338.34 bn by 2027 with a compound annual growth rate (CAGR) of 6.4% from 2019 to 2027. This market growth is owing to the implementation of embedded systems on a wide range of applications such as telecommunication, automotive, healthcare, consumer electronics, aerospace, and defense, among others. The increasing demand of Industry 4.0 solutions or automation solutions to improve manufacturing speed in plants, the adoption of 5G technology, electronic shelf label markets and telemedicine are expected to fuel the embedded automation computers market even further [5–7].

There are several major subsystems in the field of embedded automation that still share several requirements related to chip design. They must be appropriately low power, highly reliable, secure and must be programmable. The complexity of these systems is growing, as more and more applications are being integrated. Most of subsystems target application-specific hardware platforms. However, the general-purpose platforms could be manufactured in smaller quantities with less developing and fabric cost. Unfortunately, current commercial off-the-shelf (COTS) many-core processor designs cannot be used in the context of autonomous safety-related applications since safety standards (e.g ISO26262 [8] in the automotive domain) impose strict requirements that cannot be generally met with these platforms.

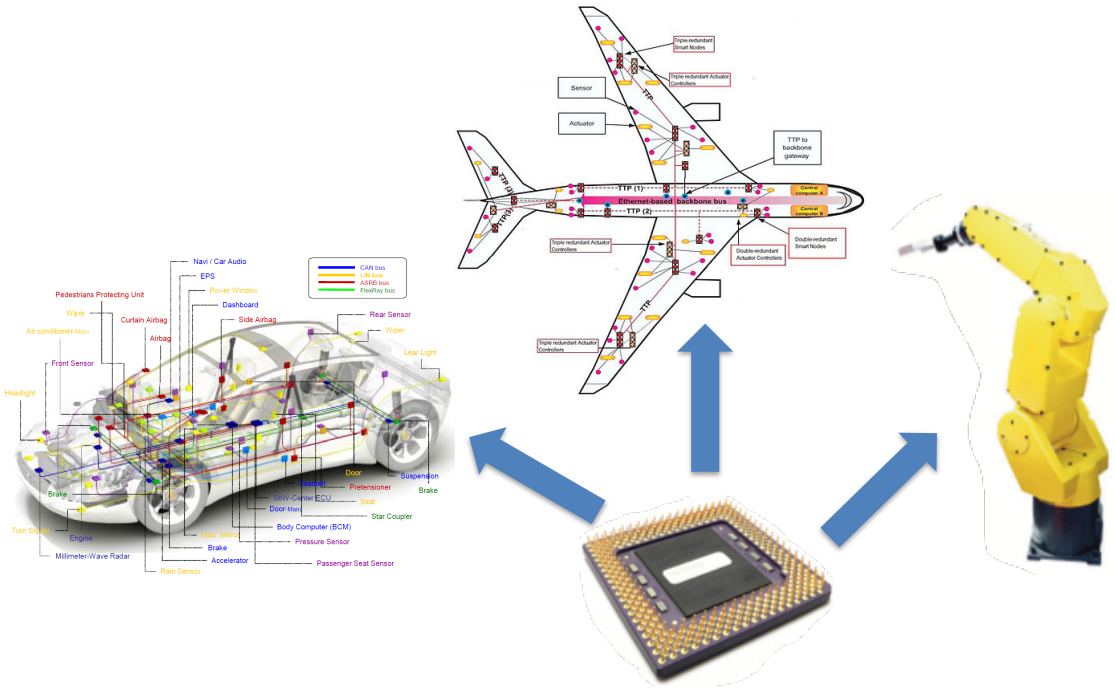


FIGURE 1.1: Different MPSoC applications in real-time systems.

Processing cores have become important in such subsystems and must sit in a system-on-chip (SoC) architecture. Several other pieces such as memory, sensors, actuators and several other pieces of IP interfaced via Network-on-Chip (NoC), the focus of this thesis, are connected to the cores.

New SoC architectures are emerging to help manage cost, power consumption and weight [9, 10]. Multi-Processor System-on-Chip (MPSoC), which combines the advantages of parallel processing with the high integration capability of SoC seems to be the most appropriate solution (see Figure 1.1).

Because of the nature of communication, the requirements imposed by different application domains (soft/hard real-time, safety critical, network topology, and so on) tend to have low data rates, small data packets, and typically require real-time capabilities, which may demand deterministic or time bounded data transfers. The need to guarantee a predictable response demands the use of appropriate scheduling schemes. However, achieving predictability in safety-critical MPSoCs is challenging due to the use of shared resources such as the system interconnect [11]. The NoC implemented in real-time MPSoCs is, therefore, impacted by this requirement and needs to be designed accordingly.

Typically, an MPSoC is composed by several IP cores and memory elements interconnected by a network-on-chip though a Network Interface (NI) module. The NoC is composed of several crossbar routers, interconnected between them and building the topology of the NoC. The network design is crucial in an MPSoC as all transmissions between cores and memories use the NoC. NoCs play an important role to enforce predictable and deterministic traffic in MPSoCs and need some effort to tackle the communication contention problem. There are three kinds of contention: source-based contention (many traffic flows from the same source at the same time), destination-based contention (many messages target the same destination at the same time), and path-based contention (many transmission flows sharing the same network resources at the same time). Bandwidth reservation is another expected property in real-time systems that needs to be taken into account by the scheduler at message injection as well as by the NoC design to enforce traffic isolation between different applications. In addition, system latency and throughput are affected by the network interconnect and the way is designed is crucial for real-time systems.

To address the communication contention problem, Time Division Multiplexing (TDM) [2, 12–15] is the most popular way to ensure time predictability in NoCs. Relying on TDM is one of the classical methods of providing configurable bandwidth reservation, guaranteed latency and throughput. TDM NoCs use slot tables that dictate resource reservations. The reservation made on the various resources ensure that communications follow their path without waiting. However, TDM-based NoCs largely ignore the fact that the application needs may change during execution, depending on its state. Besides, some works show that is possible to find a scheduling for TDM that allows achieving contention-free communications by using a computationally demanding offline process. This offline scheduling process puts serious limitations to find optimal scheduling periods for moderate NoC sizes. The use of sub-optimal scheduling periods induce performance limitations.

One way of taking into account the application state is by using a dynamic scheduler enforcing real-time guarantees. The combination of TDM NoCs with a dynamic scheduler reassigns unused TDM slots improving performance while preserving strict real-time guarantees. This technique also ensures a strong temporal isolation between communication flows needed in safety-critical MPSoCs.

The main objective of this thesis is, thus, to introduce the design and implementation of a novel and efficient time-predictable network for real-time systems in the context of MPSoC. The challenge for designing such a network is to avoid all potential message conflicts while preserving constant network message latency. To do so, conflict-free transmission is based on using a time-division multiplexing (TDM) window.

1.1 Contributions of the Thesis

In the first contribution of this thesis, we propose a new methodology based on time-division multiplexing (TDM) to eliminate packet conflicts (contention) thus, tailoring the performance bounds of the NoC to the needs of the different applications (see Figure 1.2 left side). This methodology is applied to the whole NoC design (marked with M in red circles) at right side, and allow us to map resources by time cycles to avoid packets using the same resources at the same time (conflicts). With the addition of delays at output ports at specific routers we ensure transmissions are naturally serialized and conflicts are avoided. With this methodology we build a predictable conflict-free NoC and perform a functional validation. Our NoC named DCFNoC, is able to naturally inject messages using a TDM period without the need of using a computationally demanding offline process to find the appropriate schedule. DCFNoC is suitable for mixed-criticality systems since it provides the timing isolation requirements imposed by safety-critical standards to consolidate several tasks of different criticality levels.

As a second contribution, we implement our predictable NoC in an MPSoC system (see Figure 1.2 left side). To do that, we implement both routers and NI modules and analyse the maximum attainable frequency and area overhead of implemented modules (marked with I in red circles).

As a third contribution, and in order to improve peak performance, we propose a dynamic and distributed scheduler (see Figure 1.2 left side). This scheduler builds on top of the predictable NoC to exploit its unique features. To integrate the distributed scheduler, several modules have been implemented at crossbar routers (marked with S in red circles).

As a final contribution, and for comparison purposes, we build a methodology analysis of TDM-based NoCs with real-time properties (see Figure 1.2 left side). With this

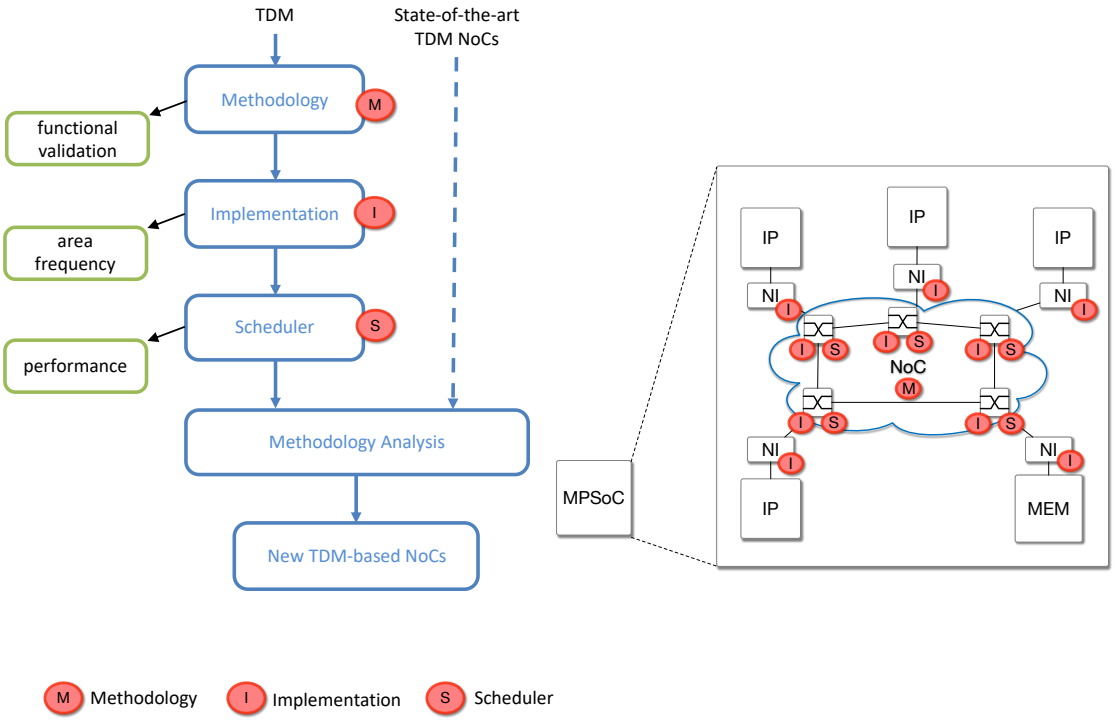


FIGURE 1.2: Main contributions to safety-critical MPSoC.

methodology we perform a comprehensive analysis to study and compare real-time NoCs trade-offs such as response time specific and implementation cost. As a result, we also derive new TDM-based NoC designs.

All in all, we can list the following specific and detailed contributions:

- Study the most recent related work in network-on-chip for real-time systems.
- Formally describe DCFNoC theory and prove that when using this NoC paradigm packet transmission is conflict-free.
- Describe a new router architecture design to apply DCFNoC theory.
- Provide timing guarantees and scalability comparison with other state-of-the art proposals.
- Evaluate the flexible bandwidth allocation property that makes DCFNoC suitable for mixed-criticality systems since it provides the timing isolation requirements imposed by safety critical standards to consolidate several tasks of different criticality levels.

- Integrate DCFNoC in a manycore processor, adjusting the design to meet determinism at application level.
- Describe how DCFNoC can be smoothly integrated in a manycore with few changes in the network interface.
- Provide performance evaluation of real workloads in the manycore to show real-time application communications behavior and thus tailoring the performance bounds of the NoC to the needs of the different applications.
- Propose a new NoC design that provides top peak performance while preserving strict real-time guarantees using a dynamic scheduler that builds on top of DCFNoC and exploits its properties.
- Describe the dynamic scheduler, evaluate the peak performance improvements over the baseline DCFNoC design and provide a scalability analysis.
- Analyse area and frequency of all modules in the new NoC design.
- Explore several TDM-based NoC solutions with real-time properties in order to compare DCFNoC with other time-predictable NoC solutions.

In summary, this thesis provides a new time-predictable NoC design paradigm where conflicts within the network are eliminated by design. The network can be smoothly integrated in the manycore system adjusting the design to meet determinism at application level. Our proposed NoC designs are suitable for mixed-criticality systems since they provide the timing isolation requirements imposed by safety critical standards to consolidate several tasks of different criticality levels.

1.2 Thesis Outline

This thesis is organized as follows: Chapter 1 has introduced the thesis, objectives, and contributions. Chapter 2 provides the necessary background on NoCs and TDM scheduling. Related work to recent Real-Time NoCs based on TDM allocation as well as COTS NoCs is discussed in this chapter. Specifically time-predictability in NoCs and systems with shared resources. Evaluation methodology and experimental platform

for implementations are also presented in this chapter. Chapter 3 describes the details of the design and implementation of the time-predictable conflict-free network on chip. Chapter 4 provides the integration of the network in a manycore processor. Chapter 5 presents the distributed dynamic scheduler that builds on top of our conflict-free network design. This scheduler is needed to improve peak performance while preserving strict real-time guarantees. Chapter 6 presents new methodologies to build time-predictable network solutions and compare them with our conflict-free network. Finally, Chapter 7 summarizes this thesis, discusses future work, and enumerates the related publications.

Chapter 2

Background, Related Work and Methodology

This chapter provides basic knowledge and the most relevant related work for NoCs and TDMA, specially real-time NoCs and COTS NoCs.

First, we introduce background concepts regarding NoCs, real-time systems and time predictability. Second, we introduce the related work on real-time NoCs with static and dynamic scheduling.

Finally, we present our design and implementation methodology, as well as methods for comparison purposes. These methods include simulation at register-transfer level (RTL), cycle accurate simulations as well as area and frequency implementation analysis.

2.1 Background

2.1.1 NoCs

A Network-On-Chip (NoC) is an interconnection network implemented inside an integrated circuit (IC), where logical nodes, known as IP cores, are interconnected each other in a System-On-Chip (SoC). NoCs main goal is to provide throughput and communication services for the chip, being efficient enough from both power and performance point of view.

The NoC uses switching mechanism, routing techniques and flow control strategies typically derived from the field of high-performance interconnection networks. Moreover, in the field of NoCs, these methods must be adapted to chip constraints which are different from traditional interconnection networks.

One important step at network design consists on determining the network topology. The topology defines the interconnection pattern between network devices and has a important impact in terms of cost and performance for the final system. This impact is due to the number of links and switches[1], its network diameter and potential number of parallel communications (as determined by the bisection bandwidth). Other aspects are related with the physical implementation which affects signal propagation latency, network clock frequency, area and power consumption.

Nowadays we can differentiate four types of network topologies [16]:

- **Shared medium networks (bus):** In this type of network all the nodes share the transmission medium and only one node is able to start transmitting at a time meanwhile the rest are receiving the information.
- **Switched media networks:** This network is formed by switches and nodes. In this type of network the nodes do not share all the switches, thus there is the possibility to transmit concurrently. We can differentiate:
  - **Direct networks:** Switches are attached at each node and these are interconnected through point-to-point links.

[1] In this thesis we use the terms switches and routers with the same meaning. Those devices connecting end nodes and building the topology.

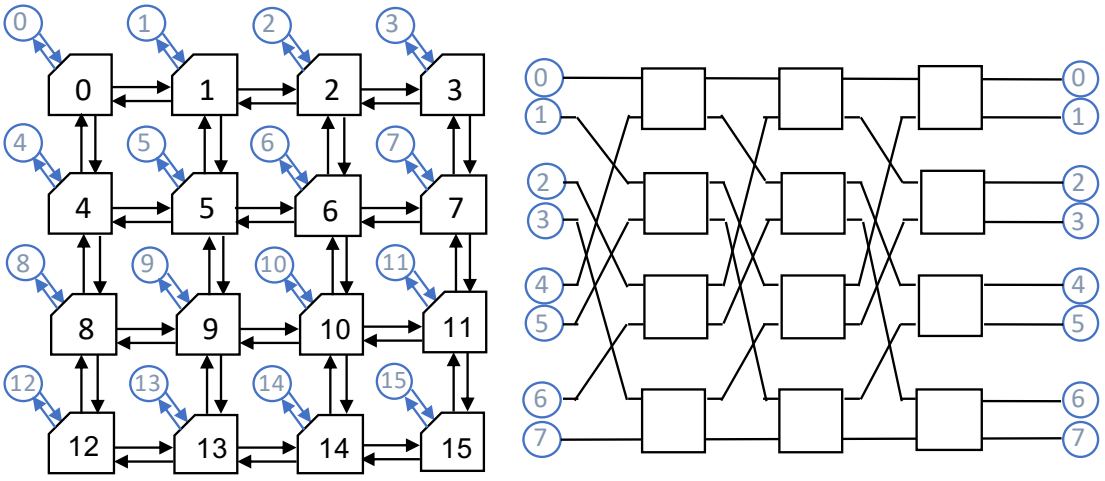


FIGURE 2.1: Direct topology at left and indirect topology at right. Nodes are represented by circles.

- **Indirect networks:** In this type of network, switches are also attached at each node and switches between them are connected through point-to-point links creating stages of switches [17].
- **Hybrid networks:** This type of network includes a combination of the previous types.

Direct networks are point-to-point networks with a regular topology where each node has a switch attached. On the other hand, indirect networks use point-to-point communication links though switches that can be connected to none, one or several nodes and also to other switches. These type of networks commonly have irregular multistage shape. Figure 2.1 shows and example of direct and indirect network. In this thesis we implement several direct networks, specifically 2D mesh networks.

NoCs must have low area overhead and be power efficient. It is important to highlight that they are usually implemented in a single 2D plane. Because of that, 2D network topologies have been common for on chip implementation, like the Tilera many-core processor family [18] and the Intel 80-core Polaris chip [19]. In fact, the 2D mesh topology is preferred due to its natural suit to a 2D surface. Link length is regular and allows a high degree of modularity. Network area increases linearly with the number of nodes. However, network bisection bandwidth increases linearly when the system size increases quadratically, which poses a system scalability problem.

2.1.1.1 Switch Architecture

The basic building block component in a NoC is the switch (or router). The switch provides communication between input and output ports following routing rules. Switches are connected to end nodes or to other switches through links. Switch architecture is determined by the switching technique that is supported and drastically affects the performance of the entire network.

Typically, a switch architecture is composed by the following elements:

- Input Buffer (IB): It temporarily stores the incoming units of information (typically called flits) from the associated incoming port and requests a routing operation to the next component.
- Routing unit (RT): Is in charge of computing the message's output port following a determined routing algorithm (see Section 2.1.1.4). At the same time, it sends a request to the virtual channel allocator (VA) and switch allocator (SA).
- Virtual channel Allocator (VA): This module arbitrates request from all input buffers and assigns one free output buffer (typically called virtual channels, see Section 2.1.1.3.1) per request.
- Switch Allocator: This arbiter grants the output port permissions to message flits. This module can be implemented to arbitrate all output ports (centralized) or instantiated per output port.
- Output crossbar (XOP): This module performs flit multiplexing for one or multiple output ports. This multiplexer is configured by the SA. Incoming flits can come from different incoming ports and from each virtual channel.

Figure 2.2 shows a baseline switch architecture with only one centralized arbitration unit [20], [21], the switch allocator. Each input port implements different input buffers, where each physical channel decomposes into several time-multiplexed logical channels or virtual channels (explained in more detail in Section 2.1.1.3.1). Each virtual channel applies individual routing decisions. As all switch buffers are centralized in only one

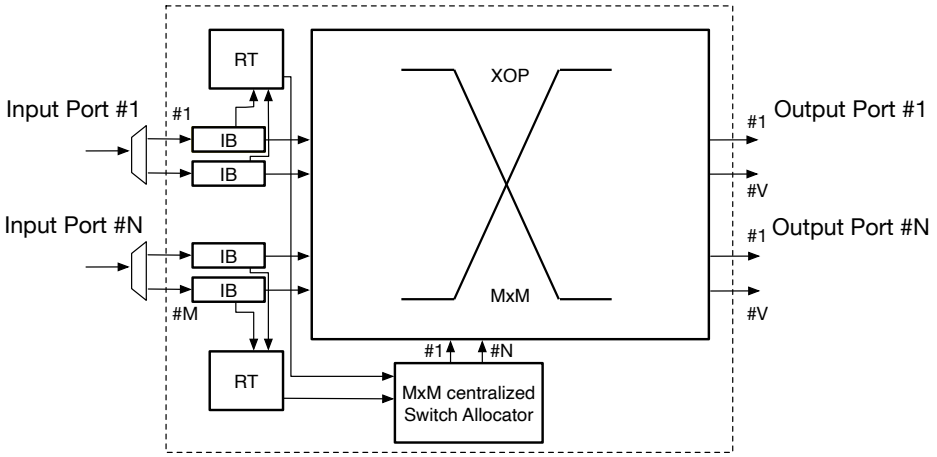


FIGURE 2.2: Baseline switch architecture. Centralized switch allocation for virtual channel switches

arbiter, the complexity of the arbiter may be high. The main drawback of a centralized arbiter is the delay.

In order to reduce the arbiter complexity, the arbitration stage may be separated using several arbiters, one per output port. In this case, every switch allocator arbitrates for free buffers to the same output port [22]. Incoming request signals may come from any incoming port RT module, thereby there is the possibility to get several incoming requests form different incoming ports and from different input buffers at the same time (at the same clock cycle). As switch allocator arbiters grant permissions for that output port, the arbiter complexity is reduced to one output port. Incoming requests, like VA arbiter, come from routing modules and thus could arrive in the same clock cycle. Moreover, SA arbiter gets requests from different incoming ports and from each one input buffers (also called virtual channels). As the output crossbar is configured by the SA associated to that output port, this multiplexer complexity is also reduced to one output port. Note that, to save area and power, buffers at output ports are usually not implemented. In this thesis we assume this router architecture as the baseline.

2.1.1.2 Flow Control

Flow control avoids loosing information between switches. To do so, there is a communication protocol between two neighbour nodes. The communication protocol establishes the flit as the minimum information unit that can be flow controlled. A message can be decomposed into packets, which incorporates routing information (therefore each packet

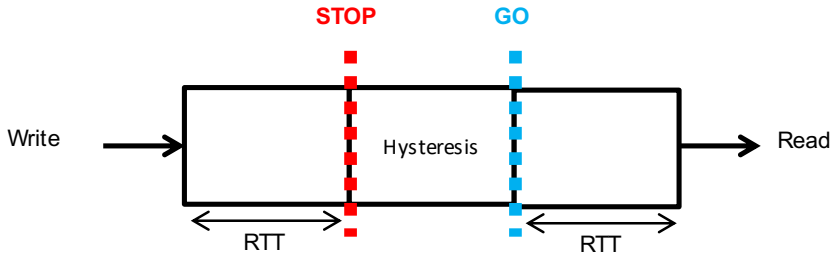


FIGURE 2.3: Stop&Go flow control.

is an independent transfer unit). Then, a message or a packet is decomposed into flits [23]. In this case, there is a header flit with routing information and the payload flits. Finally, a tail flit is also identified.

Flit size is variable and depends on the different network implementations. In fact, flit size may be equal to a complete message size (or packet) or a few bits of it. The flit is never forwarded until its storage is ensured without loss of information.

The flit transmission can be further split at the physical level into phits, representing the information unit that can be transferred in a single clock cycle through a link. Typically, in on-chip networks, flit size equals to phit size.

Flits are stored in buffers implemented at switches. Flow control techniques are in charge of determining when the flits can be forwarded according to the capacity of the buffers and the link bandwidth. There are two main flow control mechanisms: Stop&Go and credit-based. The Stop&Go mechanism is based on every receiving buffer having two thresholds to notify the sender to activate or deactivate the flow of flits. As Figure 2.3 shows, thresholds are setup taking in account the round-trip time (RTT) [24].

With credit-based flow control, each sender implements a *credit* counter for every link to indicate the number of flits that can still be stored at the buffer on the receiver side. The drawback of this flow control mechanism is the amount of credit signaling which could impact energy consumption. On the contrary, as Figure 2.4 shows, buffer size is reduced to *round-trip time*, being lower than buffer size used in Stop&Go.

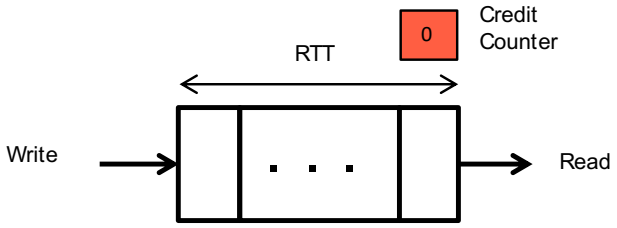


FIGURE 2.4: Credit-based flow control.

2.1.1.3 Switching

Switching techniques manage network resources in order to allocate and forward messages/packets inside the switches, while trying to minimize the message resource allocation time to maximize performance. These techniques, set connections between buffers at the input ports and output ports. Switching strategies impose several design constraints in the switch that impacts performance, manufacturing costs and power consumption of the network. Next, we describe the main switching techniques applied in NoCs:

- **Circuit Switching (CS).** This technique establishes a reserved path between source and destination node before starting messages transmission. This is achieved by injecting short control messages containing the destination of the message.
- **Store and Forward (SAF).** This strategy is performed at message granularity. When a message arrives to a switch the whole message is stored at the input port buffer before starts its transmission to the next switch.
- **Virtual Cut-Through (VCT).** In this technique messages are transmitted to the next switch when the header just arrives, and before the whole message arrives. In case the message can not be forwarded it must be fully stored in the input port buffer [25].
- **Wormhole (WH).** In wormhole switching input port buffers only have to provide enough space to store few flits, depending on the round-trip time delay, instead of the whole message. The round-trip time is the elapsed time between the information transmission and the corresponding acknowledgement is received. In WH the allocation is more efficient and, as a consequence, less power consumption in the NoC. However, wormhole may carry higher network congestion as messages can

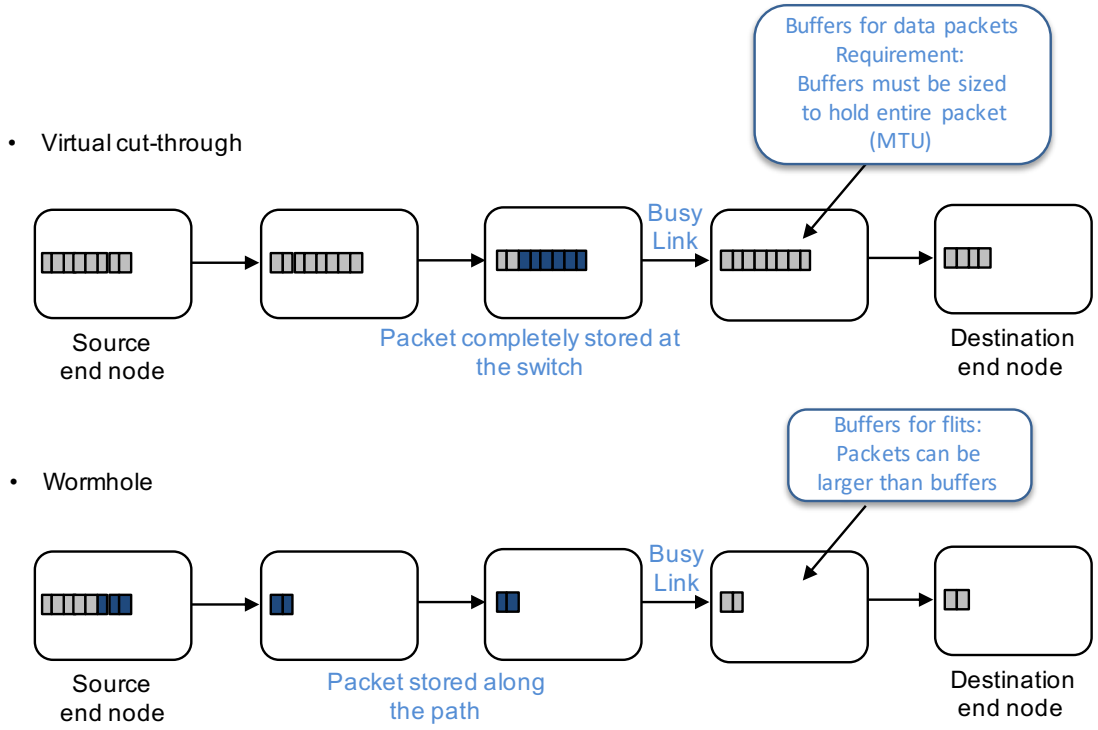


FIGURE 2.5: Virtual cut-through and Wormhole switching techniques.

be temporarily blocked, keeping several switches occupied (also called head-of-line blocking [26]). Figure 2.5 shows how VCT and WH work.

2.1.1.3.1 Virtual Channels (VC)

Virtual channels are designed to improve switching efficiency or to provide traffic isolation. Basically, the physical channel is multiplexed in several FIFO queues at input ports. Every VC is associated to a buffer and an individual flow control is performed on each VC [27]. When using VCs, if a blocked message reserves a virtual channel, other messages have the remaining virtual channels available and therefore can be forwarded using different virtual channels if the requested output port is available, see Figure 2.6. Thus, the head-of-line blocking problem is avoided. The use of virtual channels is not restricted to a specific switching mechanism and have an impact on area overhead. Originally, virtual channels have been used as a network congestion solution [28, 29], however they can also be used to improve network performance or to provide traffic isolation, avoid deadlocks in routing algorithms or prevent protocol-level deadlocks [30].

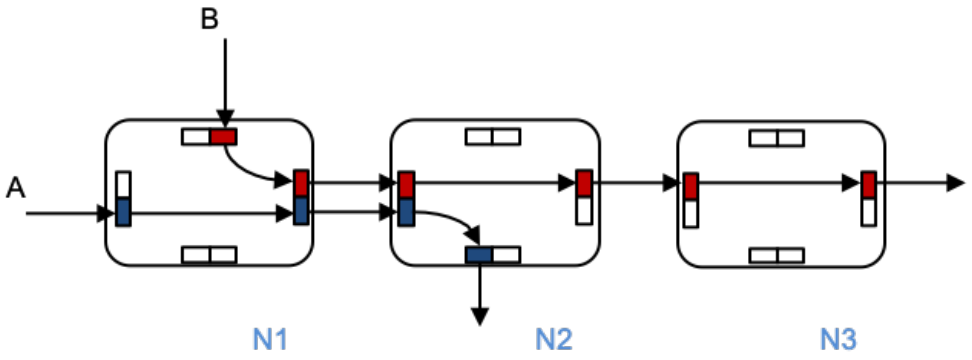


FIGURE 2.6: Virtual channels operation example.

2.1.1.4 Routing Algorithm

Routing is a fundamental aspect in a network. The routing algorithm decides which path the message has to follow to be effectively routed from its source to its destination through switches. The message will use the network resources following the switching and flow control techniques. The main goal of a good routing algorithm consists of spreading messages along the network in a well-balanced manner to minimize network congestion and using as many paths as possible. The designer must find a trade-off between efficiency, flexibility and implementation cost.

The routing algorithm can be deterministic or adaptative. In a deterministic routing a message follows always the same path from the source to the destination node, regardless of the network state (e.g. source routing). Otherwise, an adaptative routing algorithm manages to take alternative paths along the whole route in case of congested or faulty components.

A deterministic routing algorithm [23] is simple and its implementation is very efficient with low cost. Besides, message order is guaranteed, since the routing path depends only on the source and the destination node. On the other hand, an adaptative routing algorithm does not ensure in-order message delivery but offers alternative paths to increment the routing flexibility [31]. All this may lead to a higher implementation complexity and cost.

One feature to take into account in routing is the situations when messages block. We can differentiate three blocking situations: *deadlock*, *livelock* and *starvation* [28]. Deadlock is produced when the messages are blocked due to busy requested resources (buffers) and

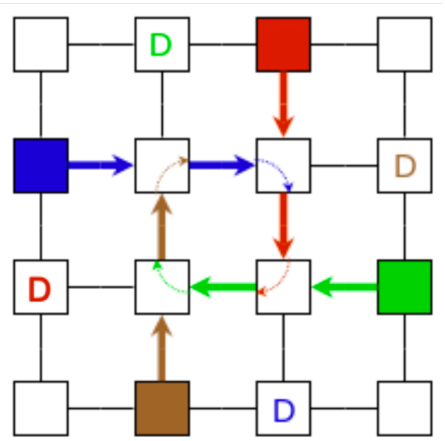


FIGURE 2.7: *Deadlock* situation due to busy requested resources (buffers) between messages in a 4×4 mesh. Each colour represents a message.

they never move forward. As Figure 2.7 shows, these messages may block others and so on until forming a cycle between them, hence they never move forward.

A routing algorithm is *deadlock-free* when its channel dependency graph is acyclic. The channel dependency graph represents all the dependencies between two channels determined by the routing algorithm. There is a dependency between two channels C_i and C_j when the algorithm requests the channel C_j for a message which is currently assigned to channel C_i. When we build a channel dependency graph and it is cycle free we can ensure that the routing algorithm is *deadlock-free* at network level. There are other algorithms that allow cycles at their channel dependency graph, and still they are free of blocking situations (for more details we recommend to read the Duato's theory [28]).

The *livelock* problem happens when a routing algorithm uses non-minimal paths and a message is always re-routed without reaching its destination. This situation may happen when the message has lower priority than others. This problem can be solved with an arbitration mechanism to avoid this situation to happen indefinitely to a message. In the same way, the *starvation* problem appears when a message is never properly routed. This situation may happen due to a lower priority to use the resources. Again, a fair arbiter for all messages can avoid this situation.

In this thesis we use the Dimension Order Routing (DOR [28]) algorithm which is deterministic and its channel dependency graph is acyclic. The DOR routing algorithm routes messages in a established dimension order. For instance, in a 2D mesh topology,

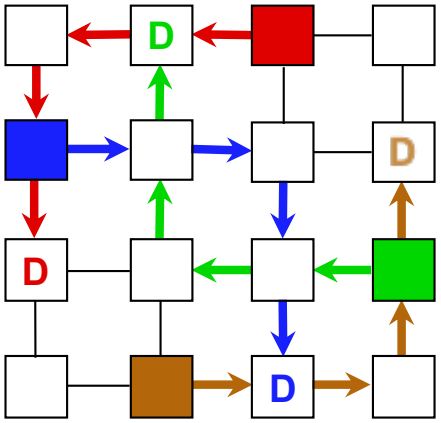


FIGURE 2.8: DOR routing algorithm in a 4×4 mesh. Each colour represents a message.

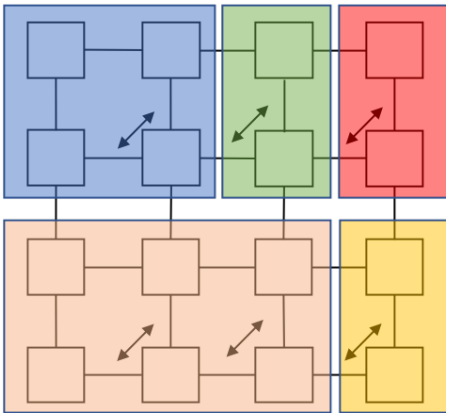


FIGURE 2.9: Segment-based routing with routing restrictions placed at partitions to avoid deadlock situations.

the routing algorithm sends messages along the X dimension until the offset in that dimension is cancelled, then the message is routed along the Y dimension. This algorithm is also known as XY routing algorithm in the context of 2D meshes. Figure 2.8 shows an example of DOR solving the blocking situation the previous last figure introduced.

A way to achieve isolation in NoCs is by using Segment-based Routing algorithm (SR) [32]. This routing algorithm is deterministic and allows to split the network in disjoint sets of interconnected switches and links called partitions. Partitions are used to avoid packets from different partitions to compete between them. In addition, some routing restrictions are placed to break potentially dependency cycles, hence its channel dependency graph is acyclic. The routing restrictions avoid packets to take some turns to preserve deadlock freedom while connectivity is kept. Figure 2.9 shows an example of SR with different partitions and the corresponding routing restrictions placed.

2.1.1.5 Arbitration

The arbitration unit is located at every switch and typically are per output port. Multiple input messages may request the same output port at the same time. In this scenario, the arbitration unit decides which message is granted to access the output port. This function can be implemented in a centralized or distributed way. If implemented in a centralized unit the arbiter inputs will be all request signals from every input port. Contrary, in a distributed way the arbiters will be scattered between the input and/or output ports [22].

A good arbitration unit must ensure the maximum match between requests and resources while ensuring fairness between all requestors. Starvation is one of the main problems in arbitration. This problem arises when some requests are never granted. An easy solution to avoid starvation is to implement a fair arbiter such as the *round-robin* arbiter.

2.1.2 Real-Time Systems

In the recent years Real-Time systems have grown in demand in the market specially in industrial environments. Real-time systems have to respond within a stipulated period of time to ensure the system functionality. Timing constraints are imposed by the real behaviour of the external world to which the system must respond or control. One of the most important features is the achievement of predictability in guaranteeing a bounded response time for software executing in these systems. Those real-time systems can be classified into four groups, depending on the accomplishment of timing restrictions.

- *Hard real-time systems.* These systems must respond after a strict set of deadlines, and missing a deadline must be handled appropriately to avoid a system failure. In critical systems, system failures may result on catastrophic consequences such as system crash or loss of human lives. These systems are also called safety-critical systems and are the focus of this thesis.
- *Firm real-time systems.* They have tolerable deadlines but it will affect the quality of the service.

- *Soft real-time systems.* These systems make an effort to reach deadlines but do not cause a high impact if a deadline is missed. Indeed, the result can still be valid to the system.
- *Non real-time systems.* The system functionality is not timing dependent.

2.1.2.1 Safety-Critical Real-Time Systems

MPSoCs are increasingly considered in safety-critical systems to cope with the high computational power demands of new applications (e.g autonomous driving systems). Unfortunately, current commercial off-the-shelf (COTS) manycore processor designs cannot be used in the context of autonomous safety-related applications since safety standards (e.g. ISO26262 [8] in the automotive domain) impose strict requirements that cannot be generally met with these platforms. Some of these restrictions are related with the response time. Ensuring predictability in MPSoCs is challenging due to its increasing functionality and complexity.

The use of manycore processor architectures was the solution for the industry to accomplish with high performance demands with better area costs. Moreover, the use of schedulers allows to maximize resource utilization while meeting application constraints. However, multi-processors can suffer from time-related delays caused by resource sharing (e.g. interconnection networks, shared caches, buses and main memory). Furthermore, potential conflicts grow quickly as the number of cores and/or the size of the NoC increase. Moreover, as interferences may occur between independent applications, they are difficult to predict and the system must be designed accordingly.

In this thesis, we enforce predictability in manycore architectures to ease their adoption in safety-critical applications.

2.1.2.2 Worst Case Execution Time (WCET) Estimates

Providing timing analysis in manycore processor architectures is challenging due to the use of shared resources. In manycores, execution time is heavily influenced by the potential interference of the applications running concurrently. Safety-critical systems require freedom from interference for SW elements integrated in the system. This translates

into being able to determine the WCET of each task that is executed in the system so that response time of the system is guaranteed to be below a given threshold. Thus, in safety-critical systems a WCET estimate is mandatory to meet the certification process requirements.

Most methods for finding WCET rely on approximations (rounding upwards when there are uncertainties) since the exact WCET is often unobtainable [33]. In the context of shared resources, an upperbound to the maximum contention suffered when accessing a particular resource is required. Research in WCET analysis focuses on reducing the overestimation to make the estimated value low enough to be valid for the design engineer.

There are two typical methods to compute shared resource contention between concurrent running tasks. In the first one, the latency suffered by a task to access a shared resource can be accounted as part of the WCET estimation process. Thus, for every shared resource we compute a latency upperbound affected by potential interferences.

In order to upperbound the impact of all potential interferences, at analysis time, every access to a shared resource is delayed its latency upperbound.

The second method of accounting for contention among accesses to shared resources is to handle contention in the schedulability analysis. This analysis is performed at system integration time, hence the exact set of tasks is known and also taking part in the system (communication flows). In order to compute the contention impact, an addition of the WCET estimates computed in isolation and the maximum contention potentially generated due to interference by the tasks running concurrently can be performed.

There are pros and cons for every method. The first suffers from over-estimation and is independent of the task [34]. The second provides tight WCET estimates, however, we need to know the tasks that will be executed at system deployment.

In this thesis we perform a maximum latency estimation at network level. Thus, for WCET estimation, we only consider the interferences of the network. WCET analysis considering other sources of execution time variability is out of the scope of this thesis.

2.1.2.3 Time Division Multiple Access (TDMA)

TDMA is one of the classical methods to avoid conflicts in shared-medium networks. This technique allows to use the same channel among multiple users by splitting access in different time slots. TDMA is used in a wide set of systems such as 2G cellular systems, GSM, satellite communications systems and passive optical networks. In automotive or avionics domains, TDMA is used in Byteflight [35] and FlexRay [36] networks.

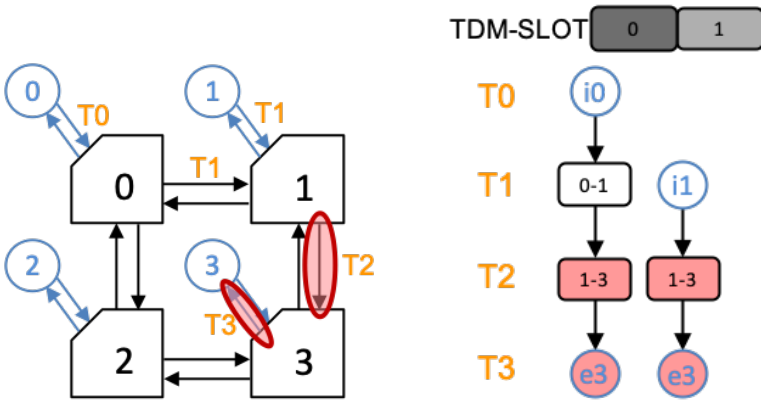
In the field of NoCs, Time Division Multiplexing (TDM) is one of the traditional methods to ensure predictability. By using TDM arbiters the switch is multiplexed in the time domain thus providing predictability. TDMA arbitration is based on a set of time slots (scheduling period or TDM window), where each slot can be assigned to a node to control injection slots.

The TDMA slot assignment targets avoiding conflicts between different communication flows at network links. The TDMA slot assignment must take into account the topology and the routing algorithm to derive conflict- free paths for every source-destination pair of end nodes. To illustrate how a conflict can affect two communication flows, lets assume a 2×2 mesh using the Dimension Order Routing DOR [6] algorithm (see Figure2.10a). Then, we assign the injection slot 0 to node 0 and slot 1 to node 1. From this topology and based on the routing algorithm we show the resulting paths from time $T0$ to time $T3$ (see Figure 2.10b). For instance, path $0 \rightarrow 3$ will cross links $\{0-1\}$, $\{1-3\}$ whereas path $1 \rightarrow 3$ will cross link $\{1-3\}$. Notice that these two paths, in red, may create conflicts since they share $\{1-3\}$ link and ejection link (red $e3$) at router 3 (ejection links are represented by $(e\#)$.

The goal of this thesis is the design of TDM-based NoC solutions with scheduling periods equal to theoretical bounds and not requiring computationally demanding scheduling processes.

2.1.2.4 Real-Time NoCs

Real-time systems impose complex constraints for interconnections networks and transmission delays must be time-bounded to guarantee a latency. In this regard, a NoC for real-time systems [37], [38] must provide guaranteed services in terms of bandwidth



(A) 2×2 mesh network. End nodes shown as circles.

(B) Conflict example for paths 0 → 3 and 1 → 3 following DOR routing and using the assigned TDMA injection slots.

FIGURE 2.10: Conflict example between different communication flows at two network links in a 2×2 mesh network using the DOR algorithm.

and end-to-end latency. A NoC with Quality of Service (QoS) has the ability to fulfill application communication requirements.

In terms of QoS we can classify NoC designs into best effort services (BE) and guaranteed services (GS) [39]. A BE NoC is assumed to provide lossless transmission without QoS support, hence lack of timing guarantees. On the other hand, GS NoC is able to ensure performance guarantees in a bounded period of time fulfilling the real-time predictability demands. Accordingly, GS NoCs are the main focus in this thesis. Next we describe the existing alternatives for real-time NoCs

Circuit-Switched (CS) NoCs are based on path reservation from the source node. Once the communication is reserved the packets start being injected. This technique requires path setup, data transmission and path release phases [40]. A NoC request latency covers from source node request till whole data transmission is completed, however CS NoCs do not provide setup latency bounds and suffer from scalability.

Packet-Switching (PS) NoCs are based on priorities using flow-control, routing algorithm and arbiters to transmit data in packets through the NoC. The priority arbitration rules and scheduling policies are used to constrain the influence of co-runner tasks interferences to simplify the WCET estimation and provide performance guarantees depending on the service priority [41], [42], [43]. Several approaches use buffers at input port switches

(virtual channels) to allow a higher priority packet to preempt a lower priority packet [44]. Virtual channels (VC) are needed to store packets blocked during its advance. VC-based NoCs are able to isolate interference between traffic flows. Priority based PS-NoCs must ensure the lack of starvation between different priority flows.

TDM-based NoCs are an alternative to achieve predictability with higher resource utilization [45]. Shared resources are associated to time tables to define the TDM period of use where each time slot is exclusively reserved for a specific transmission [46]. A global TDM scheduler coordinates TDM tables to distribute shared resources to avoid interferences. By using a TDM scheduler contention free communications through the network are guaranteed. Hence, end-to-end latencies are guaranteed as communication flows would not suffer from delays. Switch architecture is also simplified as we do not need buffers and flow control mechanism [12]. One of the major challenges in TDM NoCs is the schedule computation process to harmonize all flows among them. TDM-based NoCs are suitable for mixed-criticality systems since TDMA provides the timing isolation requirements imposed by safety critical standards to consolidate several tasks of different criticality levels.

Research in TDM-based NoCs [1] [47] shows that for the most common NoC topologies it is possible to find an scheduling for time-division multiplexing that allows achieving conflict-free communication for arbitrary traffic patterns. However, most of TDM-based approaches rely on a computationally demanding offline scheduling process. The offline process used in traditional TDM proposals puts serious limitations to find optimal scheduling periods for moderate NoC sizes [1]. Thus, TDM approaches for relatively large NoCs end up using sub-optimal solutions to find computationally affordable conflict-free schedules [47]. Regarding WCET calculus TDMA avoids tasks interferences on a shared resource, hence WCET of a task is not affected by the concurrent running tasks. The TDMA process isolates tasks providing predictability and simplifying the WCET estimation.

Other approaches combine the use of virtual channels with the time division multiplexing to provide isolation at domain level (groups of VCs). These techniques ensure contention-free communications between different domains but not between VCs in the same domain. TDM scheduling is applied at domain level. In the next section, state-of-the-art real-time NoCs and COTS NoCs are discussed.

2.2 Related Work

2.2.1 State-Of-The-Art COTS NoCs and Real-Time NoCs

Research in real-time NoCs can be classified into (1) analysis of COTS NoCs performance guarantees and (2) real-time specific NoCs. Works focusing on the analysis of the real-time properties of COTS NoCs (e.g wormhole NoCs) have shown that although performance guarantees can be achieved with these designs the achieved guarantees are generally poor when time composability aspects are also considered [48].

Many NoC proposals rely on virtual channels to ensure non-interfering operations across domains [49], [2], [50]. These solutions implement one domain buffer at each switch input port. Thus, no contention arises between different domains but only between VCs within the same domain which improves performance guarantees with respect to conventional wormhole NoC designs. In SurfNoC [49] authors implement a deterministic scheduling between domains creating a wave-like advance of messages. In PhaseNoC [2] authors propose a deterministic arbitration where a different scheduling domain is triggered in the different switch stages in order to avoid message delays along the whole path. Additionally, in [50] real-time and best effort messages are scheduled in a different way in order to provide better guaranteed throughput and fulfill application requirements.

Another existing approach to achieve predictable NoC behaviour is using virtual channel prioritization with flit-level preemption [51]. This approach achieves tight latency bounds for the highest priority flows. In general, approaches based on using VCs find limitations due to the significant amount of resources required.

Many previous real-time NoC architectures rely on time-division-multiplexing to achieve predictable message delivery. However, TDM NoCs have difficulties in finding the optimal schedules. TDM schedules can be statically [12], [13], [14], [52], [1], [53], [47], [54] or dynamically computed [55] and may be placed locally at each switch [55] for distributed routing or globally in the network interfaces (NIs) for source routing [12], [14].

The AEthereal NoC was among the first in this class of architectures: uses a virtual switching network to provide guaranteed services (GS) for performance-critical and message-switched best-effort (BE) network for applications with fewer requirements. It

uses an optimized mechanism for the static allocation of the frames which is possible thanks to the design-time knowledge of the communication requirements of target applications. AEthereal uses static distributed TDM slot tables contained in the switches that allows source routing. Aelite, a lighter version with only GS support was proposed to further simplify switches [13]. In this network routing is done through message headers and slot tables are placed at NI. A newer version of aelite called dAElite [14], provides multicast support and consequently the static routing tables are back at the switches.

In Nostrum NoC [15] TDM virtual circuits paths are fixed at design-time with variable bandwidth at run-time using the concept of looping containers. However, to fit hard real time applications, even the bandwidth must be fixed at design-time. Lu and Jantsch [52] propose a configuration technique for the Nostrum NoC allowing multiple virtual circuits to share buffers of the network. To harmonize routing tables according to global TDM schedule a backtracking search algorithm is used. In contrast, only a single assignment of a given set of virtual circuits is needed that satisfies the required bandwidth and a conflict-free operation of the NoC.

In the majority of recent proposals ([1] [53], [47] [54]) TDM schedules are allocated and configured off-line to simplify NoC hardware implementation. The theoretical minimum scheduling period for several NoC topologies and sizes are provided in [1] where an ILP formulation is provided to achieve schedules close to the theoretical minimum. However, the computational complexity of the ILP formulation makes unfeasible finding schedules for network sizes beyond 25 nodes. Thus, the approaches in [47] and [54] propose alternative optimization algorithms to find solutions also for larger NoCs. Unfortunately, this comes at the expense of periods that are significantly worse than the theoretical bound.

Communication between processing nodes is different from nodes to external memory. Since the first one follows many-to-many communication pattern, the last one requires many-to-one communication. Consequently, approaches to make many-to-one communication predictable [56], [57] are not directly comparable to the work presented in this thesis.

A solution proposed in [55] relies on two independent and parallel networks but uses them in a different way in order to achieve a distributed and dynamic resource TDM scheduler. This proposal offers a good compromise between efficiency and implementation costs for

systems with QoS requirements. This work uses credit-based end-to-end flow control to avoid data network overflow via best-effort network. These credits travel with no guarantees for the delivery delay thus can produce bandwidth underutilization and may break QoS requirements.

2.3 Methodology

2.3.1 Design Process

Computer architecture industry and academia mostly use simulation for evaluation of novel techniques and designs. Besides, considering the high cost involved in manufacturing a silicon chip in terms of time and money.

The focus of the thesis is to design and implement a novel TDM-based NoC where conflict-free transmission is based on using a time division multiplexing window. In order to evaluate how this NoC benefits safety-critical systems we need to integrate it into a manycore system. To do so, we follow several steps in the design and evaluation process. As Figure 2.11 shows the design process starts by building blocks at *block-diagram* level. Large and complex designs must be modular and hierarchical, and Verilog gives us a good framework for defining modules and their interfaces. Our entire infrastructure will be designed using verilog RTL which can be synthetized for FPGAs and ASIC.

Once hierarchy is established, the next step is the wiring of *RTL code* for modules, their interfaces and their internal functionality. To do that, we chose the Xilinx Vivado RTL simulator [58] from Xilinx company. Vivado framework performs a static syntax analysis at design time.

Once we have written the module code, we *compile* it using the hardware description language (HDL) compiler. It analyzes the code for syntax errors and also checks modules interfaces to be compatible among them. It also creates internal information that is needed for the simulator to process the design later.

In order to test the designed modules and ensure the system correctness we use the waveform provided by the Vivado simulator that simplifies the debug and functional validation stage. These modules must pass a *functional validation* test using a *testbench*.

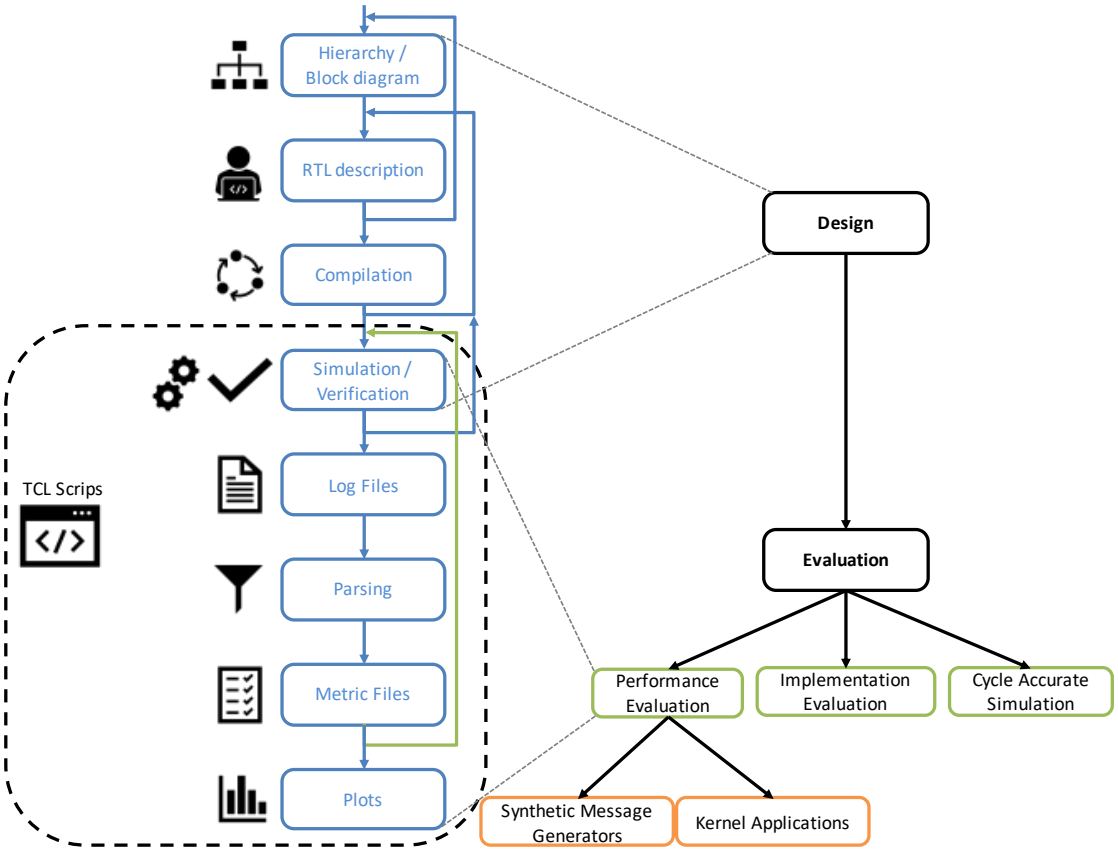


FIGURE 2.11: Design and evaluation flow.

A *testbench* provides input stimulus to the design, and to observe its outputs to check the correct behaviour.

2.3.2 Evaluation Process

Three evaluation methods have been used to analyse the designs: 1) A performance evaluation with synthetic message generators to get throughput and latency. 2) An implementation evaluation method to get maximum operating frequency and area utilization, and 3) cycle accurate simulations has been used for quick design space explorations and for comparison purposes.

2.3.2.1 Performance Evaluation

In order to analyse the system performance we need to run several simulations. To do so, we use the Vivado event-driven simulator that supports behavioral and timing simulation

for single language and mixed language designs. This simulator provides cycle accurate simulation in a waveform viewer and also supports TCL scripts, thus results would match exactly the number of cycles of a potential manycore implementation. As Figure 2.11 shows, simulations are launched by using TCL scripts coded in *bash script*.

To generate input stimulus we either use synthetic message generators to feed the network or/and launch kernel applications to stress the whole system. Simulation results are gathered in log files, later we parse the relevant information to compute evaluation metrics. Finally, plots are generated from metrics files for performance analysis.

A **Synthetic Message Generator** has been designed to perform NoC exhaustive verifications. These are implemented at each network interface. By using message generators we can test the network at different injection rates. Besides, we can define a particular message destination distribution, including uniform traffic pattern.

To evaluate the performance guarantees of the manycore we use **benchmarks from mälardalen WCET benchmarks suite** [59]. Applications in this benchmark suite have small memory footprint and thus, have very low communication requirements. Additionally, we have designed a kernel application resembling applications with high communication needs. Note that large applications cannot be effectively simulated in a detailed RTL manycore model.

To generate worst-case scenarios we combine both synthetic message generators and benchmarks. Owing to evaluate implementation results a new framework is discussed in the next section.

2.3.2.2 Implementation Evaluation

In order to infer implementation costs, we use the Cadence RC Compiler and the 45-nm Nangate library [60] to get maximum operating frequency and area utilization of our RTL designs.

By using both RTL compiler and Nangate library we get schematics and timings of our designs. Nangate library defines the standard cells used to synthesize the design. This framework allows us to evaluate our design, compare it with other existing solutions and also get some insights to improve our design either in terms of area and/or frequency.

2.3.2.3 Cycle Accurate Simulation

For comparison purposes we model our designs and state of the art TDM algorithms using cycle accurate simulations too. Cycle accurate simulations enable quicker design space exploration. Cycle accurate simulations are based on simulation models without the need of Vivado RTL simulator.

To do so, we build a simulation model of the algorithm and perform a functional verification to check behaviour correctness. It is important to check the correct behaviour to validate the simulation models to be useful. Later we build test scenarios to perform input data stimulus to the simulation models. Then, we run simulations to get evaluation results and compare designs among them.

Although cycle accurate simulations is a quick method, the main algorithm has been fully designed in verilog RTL and tested with Vivado RTL simulator.

2.3.3 PEAK Architecture

The network developed in this thesis has been be fully integrated into the PEAK manycore architecture. PEAK is the acronym Partitioned-Enabled Architecture for Kilocores developed by the Grupo de Arquitecturas Paralelas (GAP) from Universitat Politècnica de València (UPV). The aim of PEAK is to develop a fully operational manycore architecture either for research and academia purposes. PEAK has been used in some research projects such as the european project MANGO [61], besides in academia projects and its goal is to study new manycore architectures for large-scale capacity computing scenarios.

The PEAK architecture is described in Verilog RTL, is based on several identical tiles interconnected using a standard NoC (see Figure 2.12). Each tile includes a 32-bit in-order core with L1 private instruction and data caches. In every tile there is also a shared L2 cache bank, which is based on a shared and distributed organization. The coherence protocol is implemented at L1 and L2 level, using directory structures at L2 level. Core and cache memories are locally interconnected via the Network Interface (NI) module. As Figure 2.13 shows the NI provides connectivity between resources within the tile and to resources to/from other tiles.

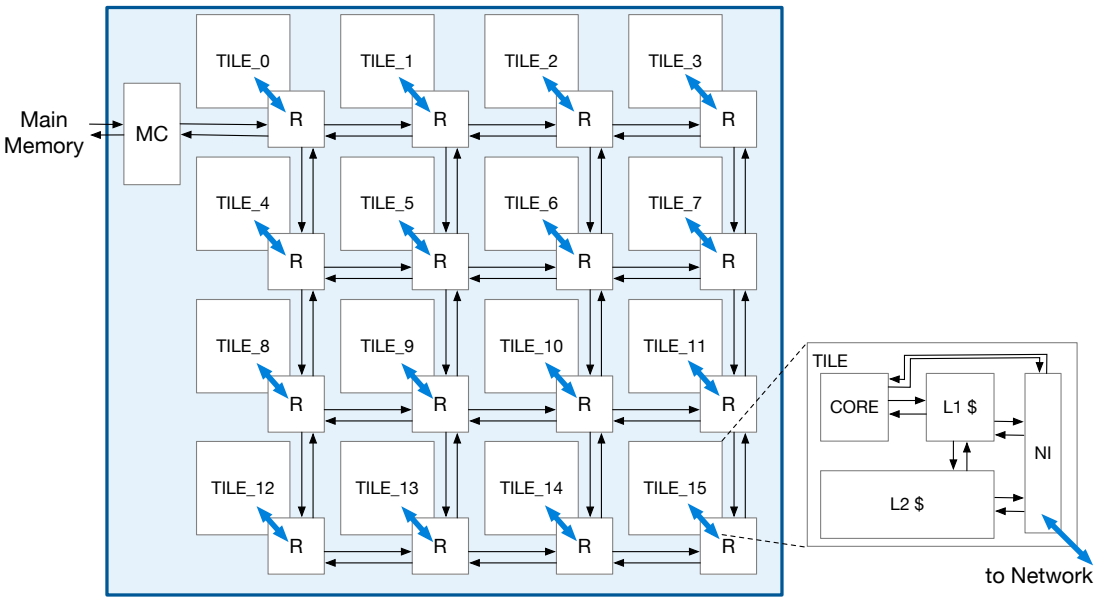


FIGURE 2.12: PEAK architecture.

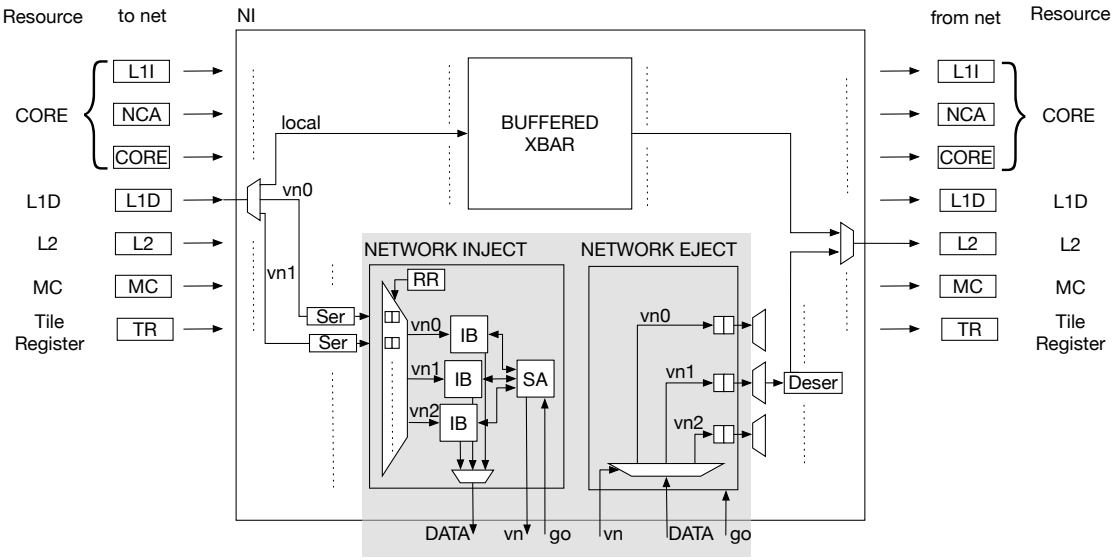


FIGURE 2.13: Network interface controller using wormhole network with virtual networks support to provide intra-tile and inter-tile resource connectivity.

The NI manages the manycore architecture communication needs. To do so, seven injector (to net) and ejector (from net) modules are defined. The core uses three injector modules L1I (instruction cache), NCA (non-cacheable addresses) and CORE (read/write to specific control registers). The remaining resources use (L1 data cache, the memory controller associated to the tile, the L2 cache bank of the tile, and the control register bank of the tile) have one additional module each. Injector and ejector are interconnected for intra-tile traffic using a buffered crossbar. For inter-tile communications,

injector modules are connected to the network inject module and serializers are used to adapt the data width on each specific case. De-serializers are used to adapt the data width from network eject modules. The network inject module implements similar logic of a router output port with virtual networks (VNs) support to separate data traffic. The network eject module demultiplexes incoming messages into corresponding virtual networks (VN).

The PEAK network implements wormhole switching with XY routing algorithm which is deadlock-free. The network routers implement virtual networks, each containing a set of virtual channels (VCs). Virtual networks enable to divide different traffic in a logic way using one input buffer for each VN supported. The network is configurable, allowing a variable number of resources and functional units. Different configurations can be created, each with a different performance/resource ratio.

In this thesis we adapt our time-predictable NoC solution to PEAK.

Chapter 3

DCFNoC: A Delayed Conflict-Free Time Division Multiplexing Network on Chip

In this chapter, we propose a NoC design in which conflict-free transmission is achieved by using a time-division multiplexing (TDM) window. However, unlike other TDM-based approaches [1] [47] [54] our delayed conflict-free NoC (DCFNoC) is able to naturally inject messages in the appropriate slot using a TDM period equal to the optimal theoretical bound. Interestingly, DCFNoC does not need a computationally demanding offline scheduling process. Note that, the offline process used in traditional TDM proposals puts serious limitations to find optimal scheduling periods for moderate NoC sizes [1]. Thus, TDM approaches for relatively large NoCs end up using sub-optimal solutions to find computationally affordable conflict-free schedules [47].

DCFNoC relies on the utilization of the channel dependency graph (CDGs) associated to the routing algorithm to identify the existing packet dependencies (potential contention) and eliminate them by the introduction of delays at strategic router output ports. The CDG helps to identify where conflicts occur in the NoC, and how these conflicts are always the consequence of dependencies between messages reaching the destination with a variable number of hops. In this context, the introduction of delays at the output ports of the routers located at particular positions in the NoC, ensures transmissions

are naturally serialized and conflicts are avoided by design. With this methodology, the network is significantly simplified since it does not need scheduling tables. Also, buffers and associated logic for flow control and arbitration within routers can be removed. DCFNoC provides the following benefits:

- **Straightforward scheduling.** Contention is simply avoided when it is enforced that no more than one node is injecting a packet in the same time slot.
- **Scalability.** DCFNoC provides better scheduling periods than competing TDM approaches and is able to find schedules in arbitrarily large NoCs.
- **Constant network message latency.** Once a message is injected into the network a path is guaranteed to reach its destination node with the same delay since all paths are forced to have the same delay.
- **Timing isolation.** Heterogeneous network bandwidth allocation can be assigned to nodes preventing bandwidth starvation or network interference.

We formalize the DCFNoC properties and prove that when using this NoC paradigm packet transmission is conflict-free. Besides, we demonstrate that DCFNoC can be obtained for any given topology and any deterministic routing algorithm. A simple end-to-end flow control is used to avoid saturation at end-points. In this chapter, we also propose a router design adapted to this new methodology.

The rest of this chapter [1] is organized as follows. First, the DCFNoC theory is formally described in Section 3.1. A general methodology for designing conflict-free networks by applying this theory is explained in Section 3.2. Next, a flexible bandwidth allocation mechanism is briefly detailed in Section 3.4. Later, a detailed router design adapted to this new methodology is provided in Section 3.6. Finally, the performance achieved by the proposed network is presented and discussed in Section 3.7.

[1] The work discussed in this chapter has been published in [62]. Sections 3.1 and 3.3 have been published in [63].

3.1 Delayed Conflict-Free Network

This section formalizes DCFNoC, a TDM-based NoC design paradigm in which conflicts are avoided by serializing message transmissions. Although DCFNoC can be used for long messages, for the sake of explanation we consider only single-flit messages. We introduce the following assumptions:

A1 *A node can generate messages targeting any other node at any rate, even broadcast messages.*

A2 *A message arriving at its destination is eventually consumed assuming an end-to-end flow control is used preventing final node buffer overflow.*

A3 *Once a message is injected into the network a path is guaranteed to reach its destination node.*

A4 *Messages are forwarded following any deterministic or partially adaptive deadlock-free routing algorithm.*

A5 *All TDM slots composing a period have the same length.*

A6 *Every router and link within the network have the same delay (we assume one cycle).*

The following definitions develop a notation for describing networks, routing functions, conflicts, and dependency graphs. A summary of notations is given in Table 3.1.

3.1.1 Definitions

Definition 1. An interconnection network I is a strongly connected multigraph defined as $I = G(N, C)$. The vertices of the multigraph N represent the set of communication nodes. The arcs of the multigraph C represent the set of communication channels. The source node of a given channel c_i is denoted as s_i and the destination node as d_i. Figure 3.1 shows a 2D mesh topology of an interconnection network.

TABLE 3.1: Summary of Notation

Sign	Description
I	interconnection network,
C	the set of channels,
N	the set of nodes,
c_{c}	a current channel,
s_{i}	source node,
d_{i}	destination node,
n_{i}	a node,
R	a routing function,
C_{l}	a channel subset,
L_{x}	an assigned layer,
M_{x}	a message,
$P(s_{\mathrm{i}}, d_{\mathrm{i}})$	a path between src to dst nodes,
$\overline{D}$	the delay of a layer or a path,
t_{m}	specific cycle time,
$\overline{C(M_{\mathrm{a}}, M_{\mathrm{b}}, t_{\mathrm{m}})}$	a conflict between two messages at a time t_{m},
CDG	a channel dependency graph,
E	the edges of CDG,
CDG_{l}	a layered CDG,
L_h	a layer of CDG_{l} in position h,
CDG_{dl}	a delayed layered CDG,
H	the network diameter,
$I(c_x)$	a injection channel,
$E(c_y)$	a ejection channel,
t_{slot}	a time slot,
P_{TDM}	a TDM period,

Definition 2. A routing function $R : C \times N \times N \rightarrow C$ provides the output channel c_y for a message located in the current node n_c at an input channel c_x and with destination node n_d:

$$R(c_x, n_{\mathrm{c}}, n_{\mathrm{d}}) = c_{\mathrm{y}} \tag{3.1}$$

Routing functions will determine the existence and severity of contention within the network as they set the communication flows. As highlighted in other works [30, 48, 64] NoC contention is a consequence of direct interference between messages, which are in turn a consequence of channel dependencies. Next, we provide a formal definition for channel dependencies.

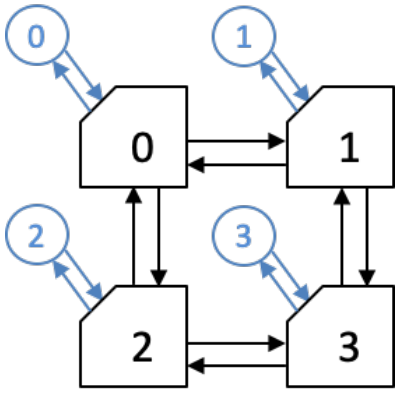


FIGURE 3.1: 2×2 mesh network. End nodes shown as circles.

Definition 3. There is a direct channel dependency from channel c_y to channel c_x, for a given interconnection network I and routing function R if c_y is needed immediately after c_x for a message located at node n_c with destination n_d.

Definition 4. A channel dependency graph CDG for a given interconnection network I and routing function R, is a directed graph, $CDG = G(C, E)$. The vertices of the graph are the channels of I and the arcs of the graph are direct channel dependencies between channels determined by R in the following way:

$$E = (c_{\mathrm{x}}, c_{\mathrm{y}}) \mid R(c_x, n_c, n_d) = c_{\mathrm{y}} \text{ for some } n \in N \tag{3.2}$$

Figure 3.2 depicts a channel dependency graph of the 2D mesh topology based on Dimension Order Routing (DOR [28]) algorithm. In this plot squares represent the vertices (channels), and the arcs represent channel dependencies. Circles represent injection and ejection channels.

Definition 5. A direct conflict $\overline{C(M_a, M_b, t_m)}$ between a pair of messages M_a and M_b for a given interconnection network I and routing function R may arise at time t_m if

$$R(c_x, n, d_a) = R(c_y, n, d_b) \text{ for some } n \in N, \tag{3.3}$$

that is, M_{a} and M_{b} are in the same node and request the same channel at the same cycle.

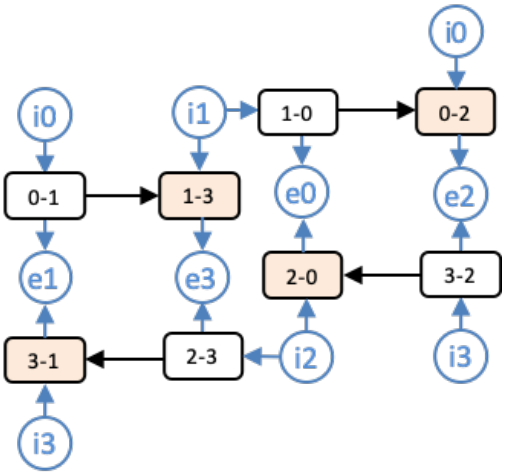


FIGURE 3.2: *CDG* for the 2×2 mesh topology and the DOR routing algorithm.

Definition 6. A layered channel dependency graph CDG_l for a given interconnection network I and routing function R, is a layered directed graph, $CDG_l = G_l(C, E)$. The vertices of the graph are the channels of I and the arcs of the graph are the channel dependencies defined by R. In contrast to CDG, all the vertices of CDG_l have an assigned layer id L_h where h represents the position of the layer. Any vertex (channel) is assigned a unique layer id. Therefore, $L_h(v)$ is a bijective function. A channel c_y with a direct dependency with channel c_x will have a higher layer id:

$$L_h(c_y) > L_h(c_x) \text{ if } R(c_x, n_c, n_d) = c_y \tag{3.4}$$

Note that in order to build a CDG_l the associated CDG must be acyclic. Therefore, the routing algorithm R must be a deterministic one or a partially adaptive one. Figure 3.3 shows the CDG_l of the 2D mesh with DOR routing. The CDG_l serves us to clearly identify potential conflicts within the network as follows. Let us assume links and routers have a delay of one cycle each and on every cycle no more than one end node injects a message in the network. If we use only dependencies not crossing a layer in the CDG_l (black arrows in the figure) then all messages will take the same amount of time to traverse the path and at every cycle there will be one message at each layer. If, however, we allow dependencies crossing layers (red arcs in the figure) to be used, then conflicts may occur at a given channel. Notice channels are located only in one layer. Indeed, the layer L_h represents the relative cycle time from injection when a specific channel along

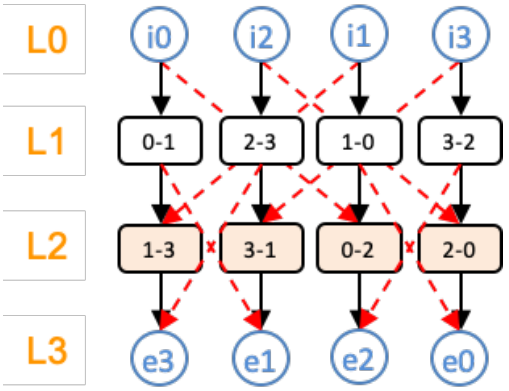


FIGURE 3.3: CDG_l obtained from CDG.

a path $P(s_i, d_i)$ is used. Assuming assumption A6 we can deduce that every layer has one cycle delay.

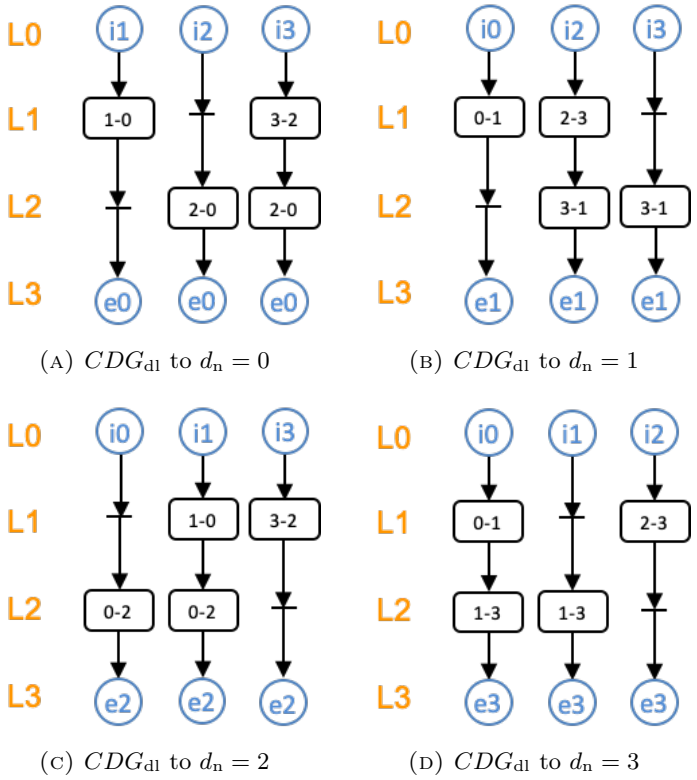
In order to remove all potential conflicts we just need to enforce two messages will not be located on the same layer at the same time (one-message-per-layer rule). To do so, we define a *delayed layered channel dependency graph*.

Definition 7. A delayed layered channel dependency graph CDG_{dl} for a given interconnection network I and routing function R, is a layered direct graph $CDG_{dl} = G_l(C, E)$, in which additional delays are introduced to remove potential conflicts. In this graph, as in CDG_l, all the vertices are assigned to a specific layer L_h. In the CDG_{dl} extra delays are introduced for channel dependencies crossing layers. Every delay is layered also in the CDG_{dl}. Therefore, all paths $P(s_i, d_i)$ have the same delay $\overline{D(P(s_i, d_i))}$.

Figure 3.4 shows parts of a CDG_{dl}. Notice that all paths have the same latency as injection channels are located at L0 and ejection channels are located at L3. Indeed, let H be the network diameter, c_i the injection channel and c_e the ejection channel. Thereby, and assuming A6, the delay for any path is:

$$\forall P(s_i, d_i) \in CDG_{dl} \quad \left\{ \overline{D(c_i, P(s_i, d_i), c_e)} \equiv L\{H\} + 2, \right. \tag{3.5}$$

In other words, for every possible path in the CDG_{dl} connecting a pair of source and destination nodes (s_i,d_i), the delay of a path is equal to the time required to traverse the set of layers corresponding to the network diameter plus two (injection and ejection layers).



(A) CDG_{dl} to $d_{\text{n}} = 0$ (B) CDG_{dl} to $d_{\text{n}} = 1$

(C) CDG_{dl} to $d_{\text{n}} = 2$ (D) CDG_{dl} to $d_{\text{n}} = 3$

FIGURE 3.4: CDG_{dl} by destination node.

Forcing all messages to traverse the same number of layers allows us to serialize transmissions and avoid conflicts. However, this requires controlling the injection. To do so, we rely on time-division multiplexing. A TDM arbiter operates by periodically repeating a schedule, with a fixed number of time slots t_{slot}. The scheduler comprises a number of slots, each corresponding to single resource access with bounded execution time in cycles.

Definition 8. A TDM period P_{TDM} is a set of time slots t_{slot}, where each slot can be assigned to a single node n_i to control injection slots, $P_{\text{TDM}} = N \times t_{\text{slot}}$.

We assume all slots have the same length, according to A5. This ensures every node can only inject messages in a given time slot and that only one node is injecting in a particular cycle.

Theorem 1. *An interconnection network I for which a CDG_{dl} can be derived is conflict-free if injection is controlled in such a way that only a message is injected in a given t_{slot}.*

Proof Sketch. We construct the proof by contradiction. Let us assume there is a conflict $C\overline{(M_a, M_b, t_m)}$ between two messages M_a and M_b. If a conflict exists then a channel c_y will be requested in the same cycle t_m by both messages. The channel will be mapped in the CDG_{dl} in a given layer $L_h(c_y)$. The distance from the injection channel to c_y is the same for both messages. Therefore, as each layer in the CDG_{dl} implies one cycle delay, the delay between injection and c_y is the same $D(M_a, c_y) = D(M_b, c_y)$. This means both messages have been injected in the same cycle time which contradicts the injection rule where only one end node can inject at a time.

The previous proof is straightforward. Indeed, assuming the existence of a CDG_{dl} it is clear conflicts are not present in the network if messages are serialized at injection time. Therefore, the complexity comes when building the CDG_{dl} for a given network I and routing function R. In the next section we provide a general algorithm for such purpose and demonstrate we can build the CDG_{dl} for deterministic and partially adaptative routing algorithms.

3.2 Building DCFNoC out of layered CDG

As an example, we assume a 3×3 mesh network using the Dimension Order Routing DOR [28] algorithm (see Figure 3.5). From the topology T and routing algorithm R we build its channel dependency graph (CDG) represented in Figure 3.6. Figure 3.6 highlights dependencies for the sink links (represented by the shaded squares). Sink links are those that do not have dependencies with others. We represent only links between routers.

The CDG can be used to derive the paths for every source-destination pair of end nodes. For instance, path $0 \rightarrow 8$ will cross links $\{0-1\}$, $\{1-2\}$, $\{2-5\}$, $\{5-8\}$ whereas path $4 \rightarrow 8$ will use links $\{4-5\}$ and $\{5-8\}$. Notice that these two paths may conflict since they share the link $\{5-8\}$ and the ejection link at router 8 (ejection links are not shown in the plot).

Our methodology to avoid conflicts in the network consists of two steps. In the first step we layer the graph to identify the critical points where conflicts may arise. In the second step, we add delays to the CDG to remove these conflicts. The computational cost of

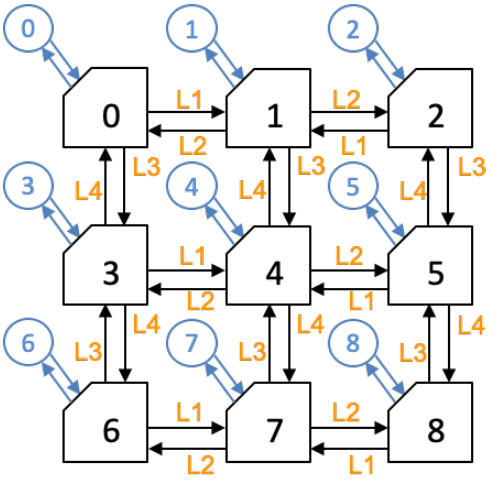


FIGURE 3.5: 3×3 mesh topology.

building the layered CDG and the modifications required to implement our DCFNoC are negligible since this exploration process is restricted and lineal to the set of valid paths provided by the routing algorithm.

Building a layered CDG. We assign a layer ($L\#$) to each node in the CDG following a partial order and we derive from the original CDG (Figure 3.6) an equivalent layered CDG (CDG_l). Notice this is always possible when the CDG is acyclic [28] and thus, DCFNoC is compatible with any deterministic or partly adaptive deadlock-free routing algorithm. We apply this ordering to the 3×3 mesh with DOR routing to label links starting from X dimension first (see how the different links are assigned to a particular layer $L\#$ in Figure 3.5). In the X dimension we label links from East To West first and from West to East later (these links correspond to layers $L1$ and $L2$). In the Y dimension we start from North to South first and from South to North later (that correspond to $L3$ and $L4$ layers). Note that in DOR every Y transitions occurs once all X transitions have been completed.

Once all links are labeled we arrange them in such a way that each network link appears only in one layer and link dependencies always go in the same direction (downwards in the figure). The number of layers is defined by the diameter of the network (i.e. the longest minimal path) plus the injection and ejection links. Figure 3.7 shows the layered channel dependency graph for our sample network design. Note that to improve readability not all possible transitions are shown in the plot. Both, injection and ejection links are also listed attached to the source and destination end nodes (represented by circles). Conflicts

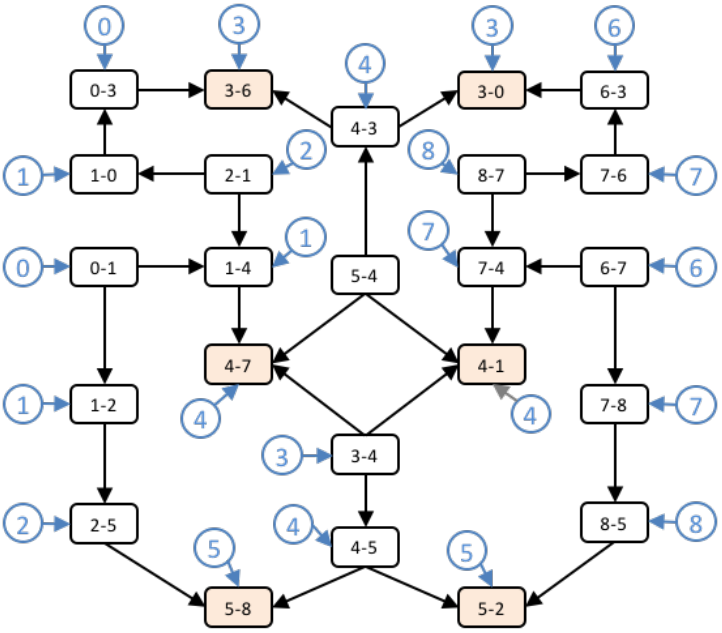


FIGURE 3.6: Channel dependency graph of the 3×3 mesh, using DOR. Squares represent links and its name indicates the routers attached to the link.

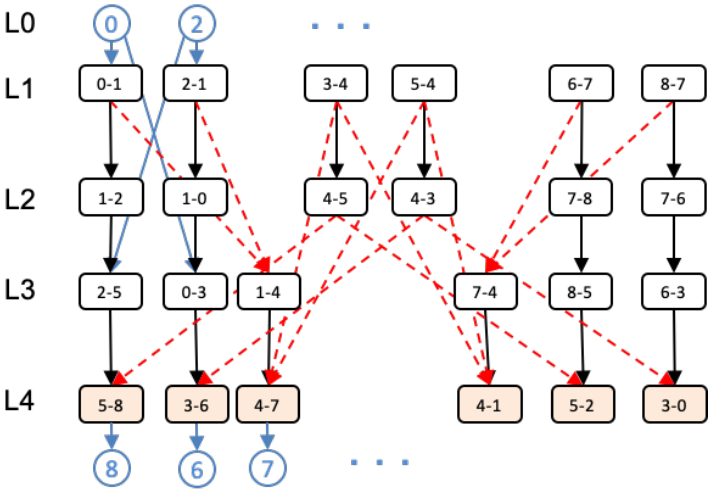


FIGURE 3.7: Layered Channel dependency graph. Potential conflicts are represented by red dashed lines.

in the CDG are identified by transitions between non consecutive layers (e.g. from $L1$ to $L3$) or, in other words, by messages targeting the same link but traversing a different number of layers. In Figure 3.7 potential conflicts are shown by red dashed lines.

Adding additional delays. To avoid conflicts, we modify the layered CDG to enforce all paths have the same delay. This is achieved by forcing all paths cross all layers in the same order (from $L0$ to $L5$). The key idea is to spread the delays among multiple routers

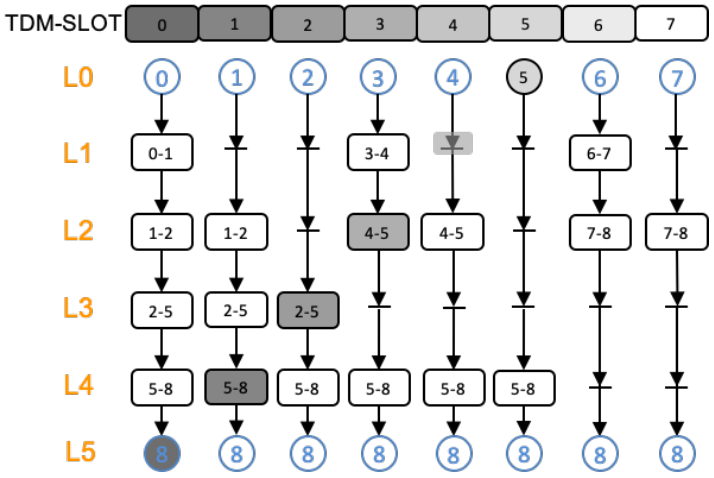


FIGURE 3.8: Layered Channel dependency graph with added delays (paths shown only from all to 8). Shades represent the TDM-slot of a potential arrangement of messages in the network.

in the path between the source node and the destination node. To do this, assuming each original link introduces one cycle delay, we add extra delays to those links that in a single hop cross more than one layer. The resulting layered CDG with added delays is shown in Figure 3.8. Extra delays are represented by a horizontal line after the arrow. Only links and dependencies for paths with destination end node 8 are shown in the plot. As shown, after this simple modification every path crosses all layers and the delay within the network is constant for all flows.

The addition of delays allows removing any potential conflict within the network by enforcing two messages will not be located on the same layer at the same time. In our NoC design TDM slots are used to guarantee every node can only inject messages in a given slot and that only one node is injecting in a particular cycle, thus enforcing the one-message-per-layer rule. Figure 3.8 shows TDM slots used by end nodes and how they use the slot to inject messages to end node 8. Six messages will be within the network, one from each end node but each message will be on a different layer. Messages from end node 0 will take four cycles as they cross four links. On the other hand, messages from end node 5 will take four cycles but only one network link will be crossed ($\{5-8\}$). Three delay cycles are added to the path to avoid conflicts.

The way in which delays are implemented is design specific. So each router will have a predefined output port delay configuration depending of its action. Notice that each delay will need a buffer to store the message. In Section 3.6 we introduce a router architecture

which can be programmed with such delays (one cycle delay per layer). In our approach, the extra delays added in CDG_l are added to the output link. Shorter paths will have additional delay cycles to serialize packets and avoid conflicts. It is important to recall that the delay for every possible path with DCFNoC is equal to the network diameter plus two (injection and ejection layers/channels).

3.3 Algorithms for the DCFNoC Design Methodology

In this section we propose the algorithm for designing TDM-based networks relying on the DCFNoC approach. The proposed methodology consists of two steps. In the first step we start from the CDG and derive the CDG_l. Then, in the second step we construct the CDG_{dl} by inserting delays at certain channels.

3.3.1 CDG_l Algorithm

The algorithm is shown in Algorithm 3.1. First, in lines 6-8 the CDG is copied to the CDG_l structure and all channels are labeled with an unassigned layer. Then, for every possible path (lines 10-27) the channels are visited and a layer id is assigned to each channel along the paths. The injection channel is assigned always to layer 0 (lines 14-15) whereas router channels are visited in the order set by the path and incremental layers are assigned (lines 19-20). Notice that channels may be already assigned by a previous path. In that case, the channel layer is inspected and if the layer is lower than the one to be assigned by the current path (lines 21-22) then an update_tree call is performed. This function searches the graph and increments the layer of channels with direct and indirect dependencies with the channel by an offset. This guarantees that the channel will have a higher layer than the previous one in the current path. In the case the channel already has a higher layer then nothing is done. After each hop, the layer, the input channel and the current node in the path are updated for the next hop (lines 23-25).

3.3.2 CDG_{dl} Algorithm

Once we have the CDG_l algorithm we proceed to obtain the CDG_{dl}. Basically we need to add delays to some channel dependencies in order to ensure every path will have the

```
 1: function build_CDGl(CDG, I, R)
 2:  path p
 3:  channel c
 4:  hop h
 5:  layer l
 6:  CDGl = CDG
 7:  for every channel in CDGl (c)
 8:    CDGl.c.layer = unassigned
 9:  end
10:  for every path (p)
11:    cx = injection_channel(p)
12:    node = p.src
13:    l = 0
14:    if (CDGl.cx.layer == unassigned)
15:      CDGl.cx.layer = l
16:    l = l + 1
17:    for every hop of p (h)
18:      cy = R(cx, node, p.dst)
19:      if (CDGl.cy.layer == unassigned)
20:         CDGl.cy.layer = l
21:      elsif (CDGl.cy.layer<=l)
22:        update_tree(CDGl, cy, l-CDGl.cy)
23:      l = CDGl.cy + 1
24:      cx = cy
25:      node = I.node.next(cy)
26:    endfor
27:  endfor
28: end function
```

ALG. 3.1: Algorithm for CDG_l

```
 1: function build_CDGdl(CDGl, R)
 3:  channel cx
 4:  channel cy
 5:  CDGdl = CDGl
 6:  for every channel in CDGdl (cx)
 7:    for every channel in CDGdl (cy)
 8:      if (R(cx, any, any) == cy)
 9:        CDGdl.arc(cx,cy).delay =
             CDGdl.cy.layer - CDGdl.cx.layer
10:      end
11:    endfor
12: end function
```

ALG. 3.2: Algorithm for CDG_{dl}

same length in time and that every path will cross all layers. To do this, we add a new field to each channel dependency (arcs in CDG, CDG_l and DCG_{dl}) representing the delay in cycles that need to be enforced as shows Algorithm 3.2. As a first action, the algorithm copies CDG_l into CDG_{dl} (line 5), then for each pair of channels c_x and c_y of CDG_{dl} (lines 6-7) check if they have a direct dependency (line 8). If so, then the delay of the output channel is set to the difference between layers of both channels (line 9).

Theorem 2. *Given an interconnection network I, and a deterministic (or partially adaptive) routing function R, the CDG_l and CDG_{dl} can always be obtained and are acyclic.*

Proof Sketch. Given R is deterministic or partially adaptive guarantees the CDG will be acyclic. Therefore, we guarantee that the algorithm used to obtain CDG_l and CDG_{dl} prevent cycles from appearing. Both CDG_l and CDG_{dl} have the same structure but only one or two new fields are added (the layer and the delay) to each arc (channel dependency). The set of edges and arcs are the same with the same configuration. Therefore, the same graph shape is inherited. Thus, CDG_l and CDG_{dl} are acyclic as well with added information. As we simply copy the CDG into CDG_l and CDG_l into CDG_{dl} then we guarantee both can always be obtained given CDG is available.

Theorem 3. *Given a path P defined from a routing function R for a network I and an associated CDG_l, the path crosses always channels in increasing layered order.*

Proof Sketch. The way the algorithm is defined guarantees a channel c_y with a direct dependency with channel c_x will have assigned a higher layer. Indeed, $L_h(c_y) = L_h(c_x) + 1$. As a path is a list of direct channel dependencies, each hop along P the channel used will have a higher layer assigned.

Theorem 4. *Given a path P for a network I, a routing function R, and an associated CDG_{dl}, the path has a delay of $2 + H$ where H is the diameter of the network.*

Proof Sketch. The path is a set of channels with direct dependencies in the CDG_{dl}. Each channel dependency has an associated delay which is the difference between layers of each channel involved in the dependency. As the depth of the CDG_{dl} is $H + 2$ the delays associated with the channel sum up $H + 2$.

3.4 Flexible Bandwidth Allocation

One of the main advantages of DCFNoC over state-of-the-art TDM approaches is that conflict-free transmission can be ensured by simply enforcing no more than one end node injects a message at each time slot. This property emanates from the fact that DCFNoC can be seen as a logical shared bus and thus, conflict-free message transmission can be ensured by simply enforcing the atomic utilization of time slots. This property can be exploited to implement heterogeneous bandwidth allocation schemes across end nodes to accommodate the communication requirements of the different applications running in the system. For instance, heterogeneous bandwidth allocation is a desirable NoC feature

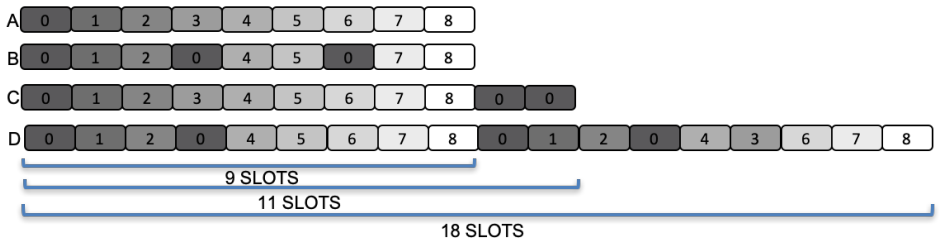


FIGURE 3.9: Different bandwidth allocation options. Each box represents an injection slot and each label indicates the end node the slot is assigned to.

in the context of automotive applications. Automotive applications using AUTOSAR are composed of several runnables that can be executed in parallel and have different computing and communication requirements [65].

Heterogeneous bandwidth allocation can be easily implemented in DCFNoC at the edges. Figure 3.9 shows four potential allocation windows (A, B, C, and D) in a 3×3 NoC. The first allocation (A) is the one corresponding to an homogeneous bandwidth allocation strategy in which all nodes get the same bandwidth (1/9). Example B shows the case in which nodes 3 and 6 are inactive and this bandwidth is assigned to end node 0 that gets 3/9 of performance guarantees. Example C shows how increasing the period from 9 to 11 can be used to assign node 0 3/11 of the total bandwidth while the rest get 1/11. Finally, example D shows a period of 18 cycles in which node 0 gets 4/18, and nodes 3 and 5 get 1/18 each. Each of the rest of end nodes get 2/18 of the total bandwidth. In general, DCFNoC allows using fine-grained bandwidth allocation to match different applications needs.

DCFNoC is application agnostic and can be configured to fit the application bandwidth requirements. Indeed, a profile of the application is usually obtained and the network is configured to adapt to the traffic requirements between end nodes. Each end node is then configured with some assigned slots which lets the node to achieve a certain bandwidth of the network.

3.5 Broadcast Support

One significant advantage of DCFNoC is its natural broadcast support. DCFNoC is able to support broadcast by simply adding a broadcast bit and forwarding messages at every

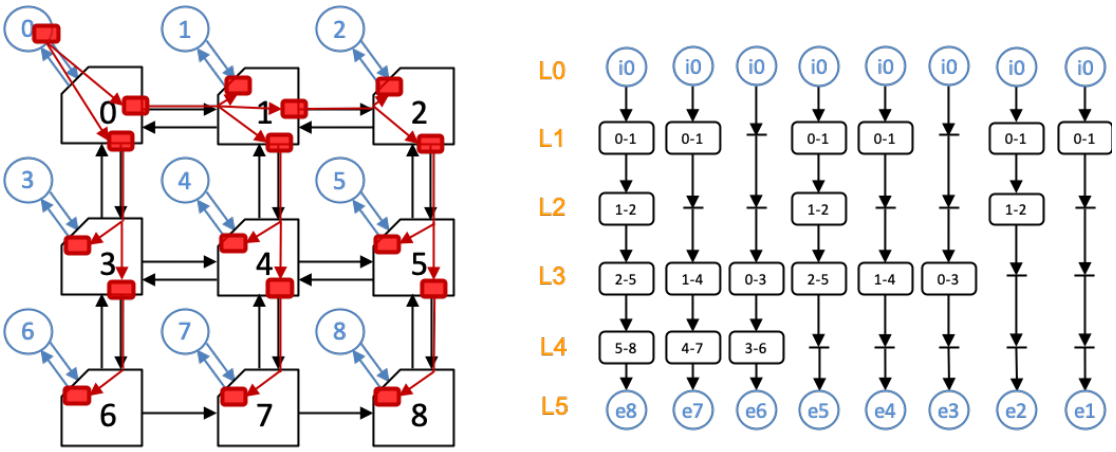


FIGURE 3.10: Broadcast support using the DOR routing algorithm in a 3×3 mesh (paths shown only from 0 to all). Both, injection and ejection end nodes are represented by circles and channels are represented by squares.

input port following DOR routing algorithm, see Figure 3.10 (left side). A message that is injected into the network will be forwarded without any stop following the delays and links imposed at design time. Broadcast messages duplicate when needed to reach all destinations. However, every message copy follows the same delay approach using the latches and therefore, all the paths have the same latency, see Figure 3.10 (right side).

DCFNoC broadcast support is used to avoid wasting a complete TDM window to broadcast to all nodes. Thus, every node will broadcast its message on a single slot to the rest of nodes. Also, DCFNoC guarantees all broadcast messages arrive at the same time to all nodes which simplifies the scheduler design. Later, we will see how broadcast can be used to efficiently schedule transmissions of messages.

3.6 Router Design

The structure of the DCFNoC router is shown in Figure 3.11. This router consists of multiplexers, registers and OR gates. This simple design leads to low resource utilization, high frequency, and low power consumption as we will show in Section 3.7.4.

The router implements five ports and the DOR routing algorithm. Messages at input ports are routed and latched at the corresponding output port without needing any arbitration logic nor flow control. However, the router supports the case of receiving multiple conflict-free messages through different ports at the same time, and also several

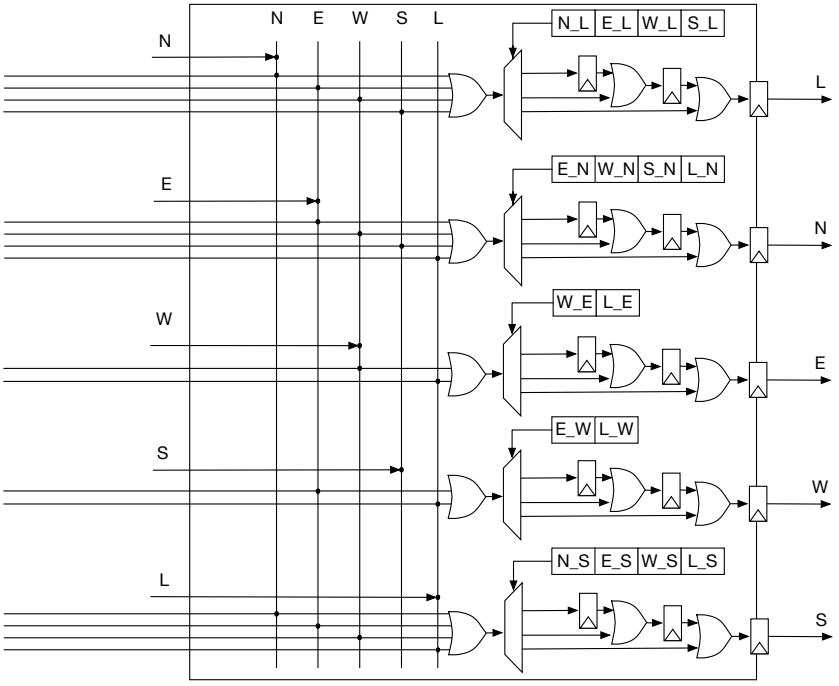


FIGURE 3.11: DCFNoC router input/output ports connections with output delay registers.

of them targeting the same output port but targeting different delay latches. Each output port implements a de-multiplexer with several single-cycle delay latches. A registered configuration vector programs the de-multiplexer on each output port. Input port arrival ID is used to index the configuration register and set appropriately the de-multiplexer. Notice that depending on the routing algorithm the configuration register may have a varying number of slots, from two to four when DOR is used. This depends on the maximum number of output dependencies of a given link accounted in the CDG. Notice that each configuration slot will impose a varying number of delay cycles to the message pipeline transmission. This will enable the proper appliance of the CDG_{dl} methodology.

As an example, Figure 3.12 shows delays introduced by DCFNoC for paths $0 \rightarrow 3$ and $2 \rightarrow 3$ in a 2×2 mesh following the example provided in Figure 3.4d. As we can see in the plot, the latency for both paths is three cycles since they traverse the same number of latches. Both, injection and ejection end nodes are also shown (represented by circles). Note that the extra cycle delay of path 2 to 3 will be set at output port *local* of router 3. In a $N \times M$ Mesh, the maximum number of extra cycle delays implemented in each output port is $(N-1)+(M-1)-1$.

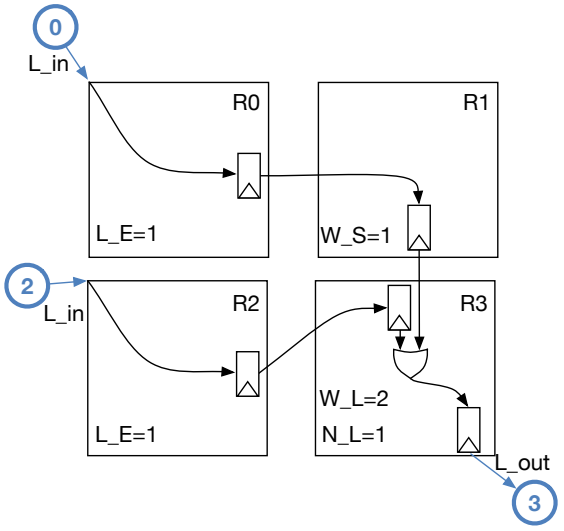


FIGURE 3.12: DCFNoC mesh with output delay registers for paths $0 \rightarrow 3$ and $2 \rightarrow 3$.

3.7 Performance Evaluation

In this section we compare the performance guarantees provided by DCFNoC with the ones provided by similar TDM approaches. Additionally, to analyze the feasibility of implementing the proposed NoC we provide area and maximum attainable clock frequency and compare these numbers with the ones obtained with an standard wormhole router [28] [29].

3.7.1 Experimental Setup

DCFNoC is described in verilog RTL and can be synthetized in FPGAs and ASIC. We have simulated DCFNoC using the Xilinx Vivado [58] RTL simulator. DCFNoC configuration implemented uses 64-bit width links mesh network. The network is feed by a message system generator implemented at each network interface using uniform traffic pattern. For comparison purposes we model DCFNoC and state of the art TDM algorithms. Maximum operating frequency and area utilization are obtained using Cadence RC Compiler and the 45-nm Nangate library [60]. The wormhole NoC implemented is also 64-bit width and uses 8 slot input buffers with Stop&Go flow control.

TABLE 3.2: A comparison of schedule length and network latency.

Top.	Size	Period				
		TLB [1]	ILP [1]	SYM [20]	PhaseNoC [2]	DCFNoC
Mesh	2×2	3	5	-	4	4
	3×3	8	10	28	9	9
	4×4	16	18	59	16	16
	5×5	32	34	112	25	25
	6×6	-	-	-	36	36
	7×7	-	-	-	49	49
	8×8	-	-	481	64	64

3.7.2 Timing Guarantees

For comparison purposes we model DCFNoC and state of the art TDM algorithms. Table 3.2 shows the scheduling periods of our approach and compares them with the ones obtained by other state-of-the-art proposals. As shown in the table our NoC design achieves the smallest scheduling periods in all configurations. Our approach is able to improve the period of the ILP-based scheduling [1], an approach that is able to find computationally viable schedules for meshes up to 25 nodes. Authors in [1], also formulate a minimum period based on theoretical lower bounds. Which is almost equal to our schedule period. This value is represented in Table 3.2 as TLB. The improvement in period achieved by our approach w.r.t [1] for a 5×5 mesh is 26.47%. When compared with SYM [47], which is able to find schedules for larger NoCs, our approach reaches a 77.67% improvement for a 5×5 mesh. PhaseNoC [2] has a TDM period equal to our schedule period but this network requires one domain buffer at each router input port incurring in additional area overheads and additional latency at every hop.

Figure 3.13 shows latency results of DCFNoC compared with the optimal ILP schedule proposed in [1] and PhaseNoC [2] for a 5×5 NoC for different message injection rates. Latency results are computed for randomly generated messages considering that messages can only be injected in the network in their assigned slot. Latency results shown in this plot represent end-to-end latency values. Additionally, for DCFNoC the latency experienced by the messages once injected in the network is the same for all nodes regardless the target destination since all messages always experience the same latency.

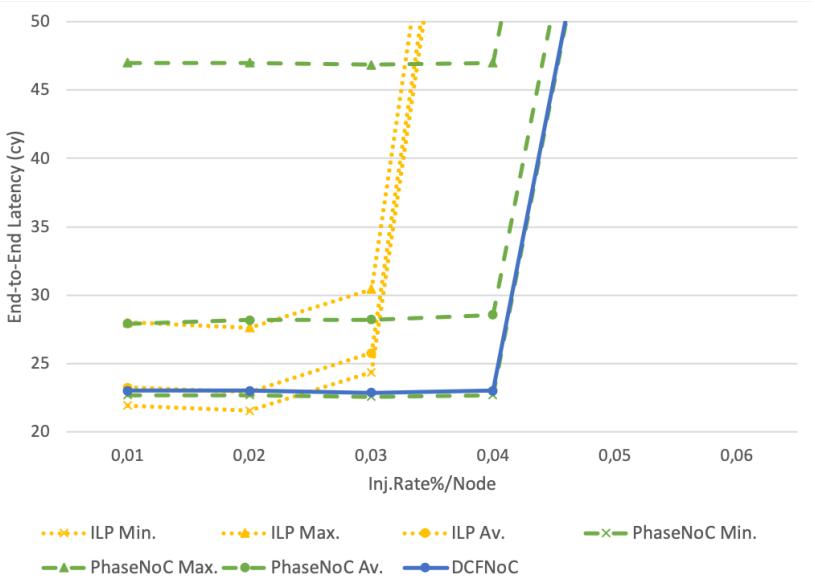


FIGURE 3.13: End-to-End latency of DCFNoC vs ILP [1] and PhaseNoC [2]. Y axes starts at 20 to improve the visibility of the comparison.

On the contrary, in ILP [1] and PhaseNoC [2] the latency experienced once a message is injected depends on the amount of hops each of the communication flow traverses. Thus, in the figure we show values for the *min* and *max* that represent the flow traversing the lower and the larger number of hops. The result of averaging out all potential flows is represented as *average*. We do not provide results for SYM [47] because scheduling periods for this implementation are much worse than the ones achieved by our proposal.

Figure 3.14 shows a scalability comparison of DCFNoC compared with the optimal ILP schedule proposed in [1] and PhaseNoC [2] for different NoC sizes. As shown in Figure 3.14 for very small injection rates and the smallest NoC sizes DCFNoC latency is slightly worse than the one achieved in ILP [1] for the shortest paths but better for the longest ones. Note that for a 3×3 mesh PhaseNoC and DCFNoC have nearly the same latency. For higher NoC sizes and/or higher injection rates DCFNoC achieves always better results. The reason for this is the smaller period of DCFNoC that decreases the average time each message is waiting until it is aligned with the assigned slot. The smaller period also enables DCFNoC achieve small latency values for higher injection rates. Although PhaseNoC [2] and DCFNoC have the same schedule period, the latency of PhaseNoC is higher. As we can see in Figure 3.14 DCFNoC has better scalability than other state of the art TDM proposals for large NoC sizes.

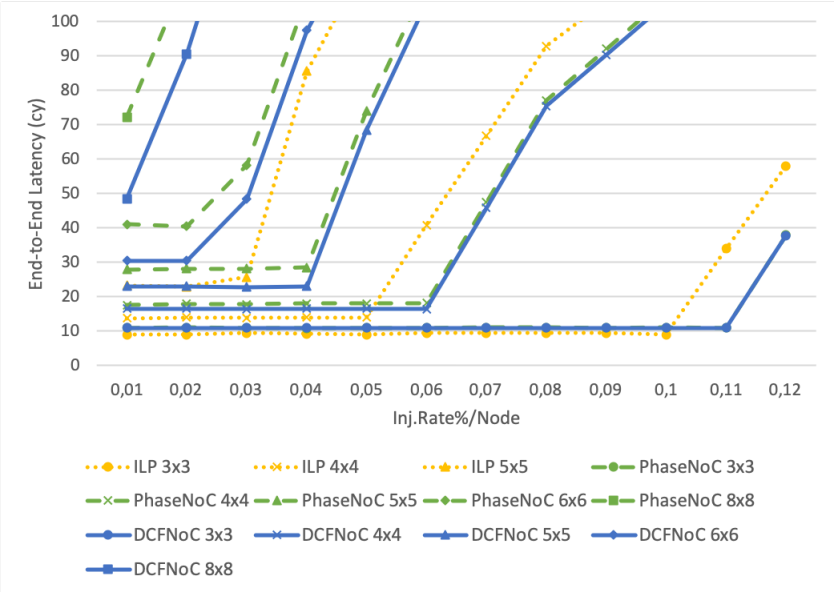


FIGURE 3.14: Scalability of DCFNoC vs ILP [1] and PhaseNoC [2].

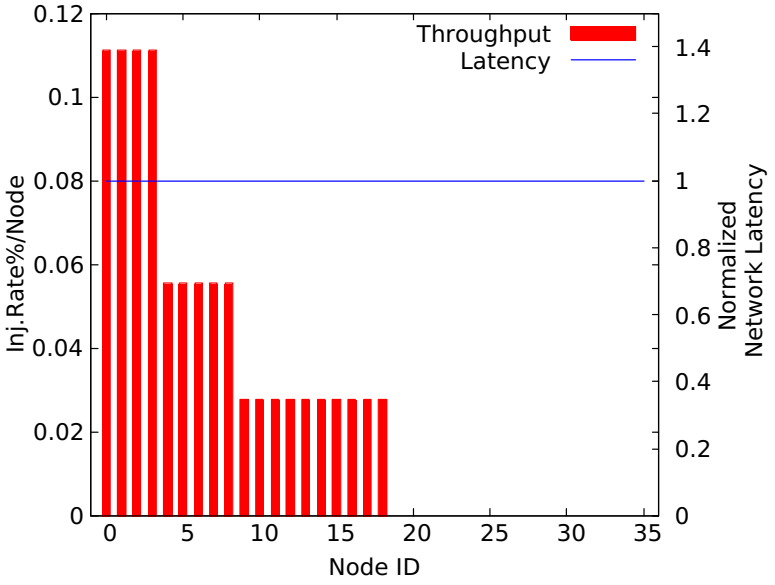


FIGURE 3.15: Heterogeneous bandwidth guarantees assignment in a 6×6 mesh.

3.7.3 Flexible Bandwidth Allocation

In order to test the capabilities of the flexible bandwidth allocation of DCFNoC, we perform an experiment on a (6×6) mesh using uniform traffic. At the network edges we statically allocate 36 1-cycle time slots to nodes. One time slot is equal to 1/36 of total bandwidth, resulting to a 2.7%. In particular, $(0-3)$ nodes get assigned four time slots each, $(4-8)$ nodes get two slots, and $(9-18)$ nodes get one slot. Remaining nodes do not inject traffic.

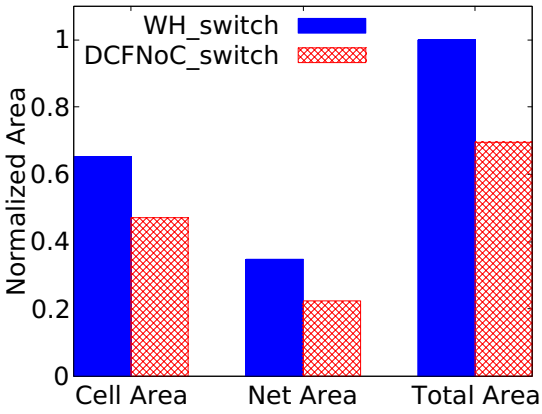


FIGURE 3.16: Area overhead.

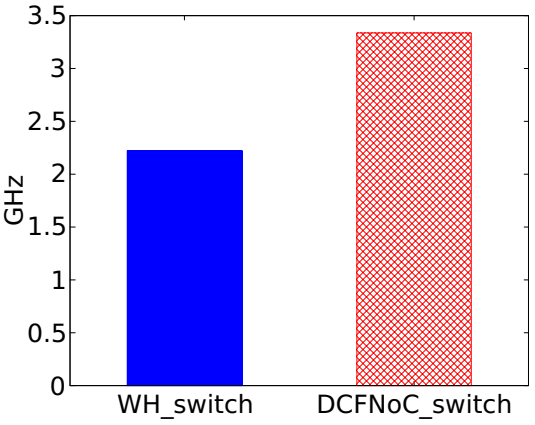


FIGURE 3.17: Maximum attainable clock frequency.

Figure 3.15 shows how bandwidth of some end nodes is improved (nodes from 0 to 8) at the expense of the bandwidth of other nodes. Note also the fact that latency of messages once they are injected in the NoC is kept constant.

3.7.4 Area and Frequency

For the implementation results we consider a DCFNoC router for a 8×8 mesh. The wormhole router (WH) used for comparison purposes implements one single virtual channel, round-robin arbitration, and XY routing. We rely on this canonical and pipelined router implementation since it is one of the most common routers for on-chip networks. By comparing with this wormhole router we have a clear reference of which is the area overhead and maximum attainable frequency.

Figure 3.16 shows area overheads for the two routers when targeting high frequency. The DCFNoC router uses 10.21% less cells than the WH-based one. As a consequence, we obtain total area savings of 30.42%. The WH router needs additional logic in order to implement input buffers, flow control logic, routing units, output port arbiters and crossbar interconnect. On the other hand, the DCFNoC router implements very simple routing logic in order to compute the output port, a crossbar interconnect as well as output delay registers.

We have also analyzed the maximum attainable clock frequency by each router. As a first insight, the DCFNoC router is a one-cycle delay router while the wormhole router is a 4-stage pipelined router. Figure 3.17 shows, as expected, the simpler DCFNoC router

design gets a significative boost in clock frequency by improving wormhole router's one by 50%. The critical path of the wormhole router limits clock frequency to 2.22GHz. However, the DCFNoC router exhibits a critical path of 300 ps leading to a clock frequency of 3.33 GHz.

3.8 Summary

In this chapter we have proposed a new methodology for NoC design based on the CDGs theory, which guarantees by design the avoidance of contention within the NoC. The proposed approach, DCFNoC paradigm, improves over state-of-the-art TDM proposals by achieving scheduling periods that almost match the theoretical lower bound. While traditional approaches have difficulties to find schedules for large networks DCFNoC is able to find conflict-free scenarios in arbitrarily large NoC sizes without degrading the quality of the achieved guarantees. Finally, we have also shown the feasibility of the proposed approach by implementing a high-speed router design with very small area needs.

However, as possibly deduced from the chapter, our solution limits network throughput by enforcing one message injection at a time. In following chapters we will address this issue. In the next chapter we adapt our solution to PEAK, a manycore architecture.

Chapter 4

Enforcing Predictability of manycores with DCFNoC

Due to the high number of cores competing for the shared resources, the network-on-chip (NoC) becomes a key resource with a high influence in the experienced contention. For instance, wormhole NoCs implemented in COTS manycores have been shown to introduce a huge negative impact in the quality of WCET estimates [48].

On the contrary, DCFNoC, as described in the previous chapter, provides the timing isolation properties required by safety critical standards to consolidate several tasks of different criticality levels. Additionally, the good properties of DCFNoC allow easily allocating heterogeneous bandwidth guarantees to the different tasks executed in the MPSoC, thus, tailoring the performance bounds of the NoC to the needs of the different applications while network interference and bandwidth starvation are avoided.

In this work, we show how DCFNoC can be used to control interference in manycore processors. We show how DCFNoC can be smoothly integrated in the manycore system when an appropriate network interface is designed or modified. Finally, we show performance guarantees achieved with our DCFNoC are much superior to the ones provided by a baseline default wormhole NoC.

This chapter [1] is organized as follows. First, the DCFNoC integration in a manycore system is described in Section 4.1. Next, a performance analysis of the proposed DCFNoC

[1] The work discussed in this chapter has been published in [63].

manycore integration without losing its time predictability property is studied in Section 4.2.3. Later, a performance evaluation of real workloads is discussed in Section 4.2.5. Finally, area overhead and maximum attainable frequency are analysed in Section 4.2.6.

4.1 Integrating DCFNoC into a Manycore Design

We start from an existing manycore processor architecture as the one depicted in Figure 4.1. The design, described in Verilog RTL, is based on several identical tiles interconnected using a standard NoC.

4.1.1 Tile Architecture

Each tile includes a 32-bit in-order core with L1 private instruction and data caches. A shared L2 cache bank is included also on each tile. All the L2 cache banks from all tiles form the L2 cache of the manycore. To keep data coherent, a coherence protocol is implemented both at L1 and L2 cache levels. The coherence protocol relies on directory structures at L2 level. Both the core and cache memories are interconnected via the Network Interface (NI) module, which provides connectivity between resources within the tile and to resources to/from other tiles. The NI is connected to a router which, in turn, is connected to routers of neighbouring tiles, building a 2D mesh topology.

4.1.2 Network Interface

The manycore architecture has a wide variety of communication needs. Indeed, memory requests are triggered by the cores as well as coherence requests between memory resources are exchanged. In addition, debug and monitoring information is communicated between the resources. To deal with this communication overhead and complexity, a sophisticated NI is used, depicted in Figure 4.2 (top part).

Seven injector (to net) and ejector (from net) modules are defined. The core uses three modules: L1I (instruction cache), NCA (non-cacheable addresses) and CORE (read-/write to specific control registers). The remaining resources (L1 data cache, the memory

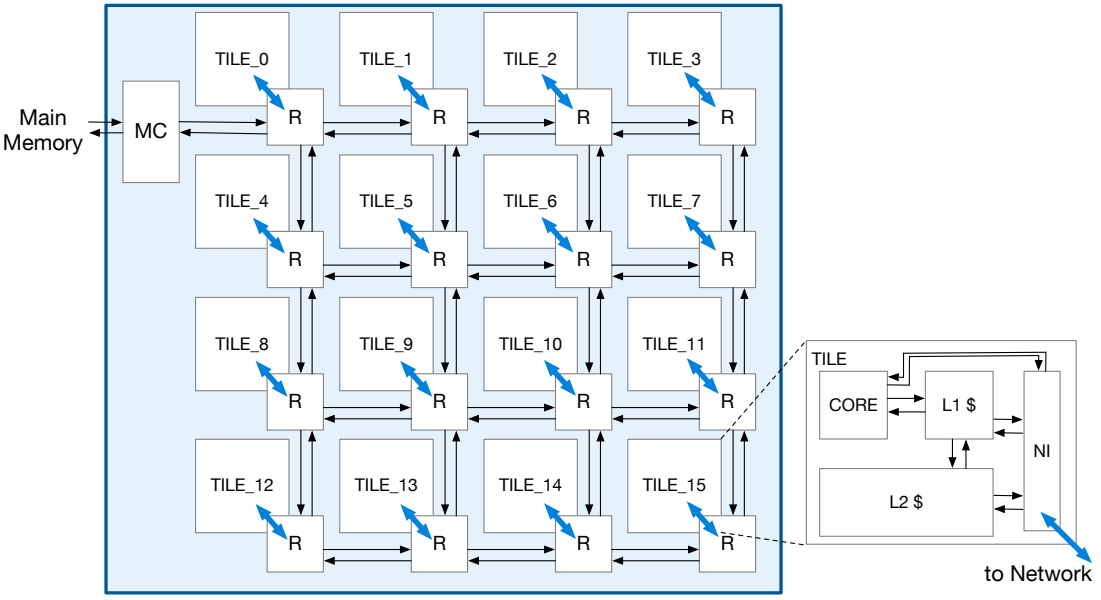


FIGURE 4.1: Baseline manycore architecture implementing a configurable number of virtual networks to separate data traffic.

controller associated to the tile, the L2 cache bank of the tile, and the control register bank of the tile) have one additional module each.

The injector modules are connected both to the ejector modules (for intra-tile traffic) and to the network inject module (for inter-tile traffic). In the case of inter-tile traffic, serializers are used to adapt the data width on each specific case. Ejector modules are connected also in a similar way to injector modules and to the network eject module. De-serialisers are used to adapt the data width of the network.

The network inject module implements similar logic of a router output port. In the manycore architecture the routers implement virtual networks (VNs) to separate data traffic. A multiplexer separates every input in virtual networks. One input buffer is used for each VN supported. A switch allocator (SA) module is used to assign network resources to messages and to grant access to the eject link. The network eject module is much simpler as it only demultiplexes incoming messages from the network into the corresponding virtual network (VN). A two slot buffer is used on each VN at eject in order to guarantee 100% network throughput.

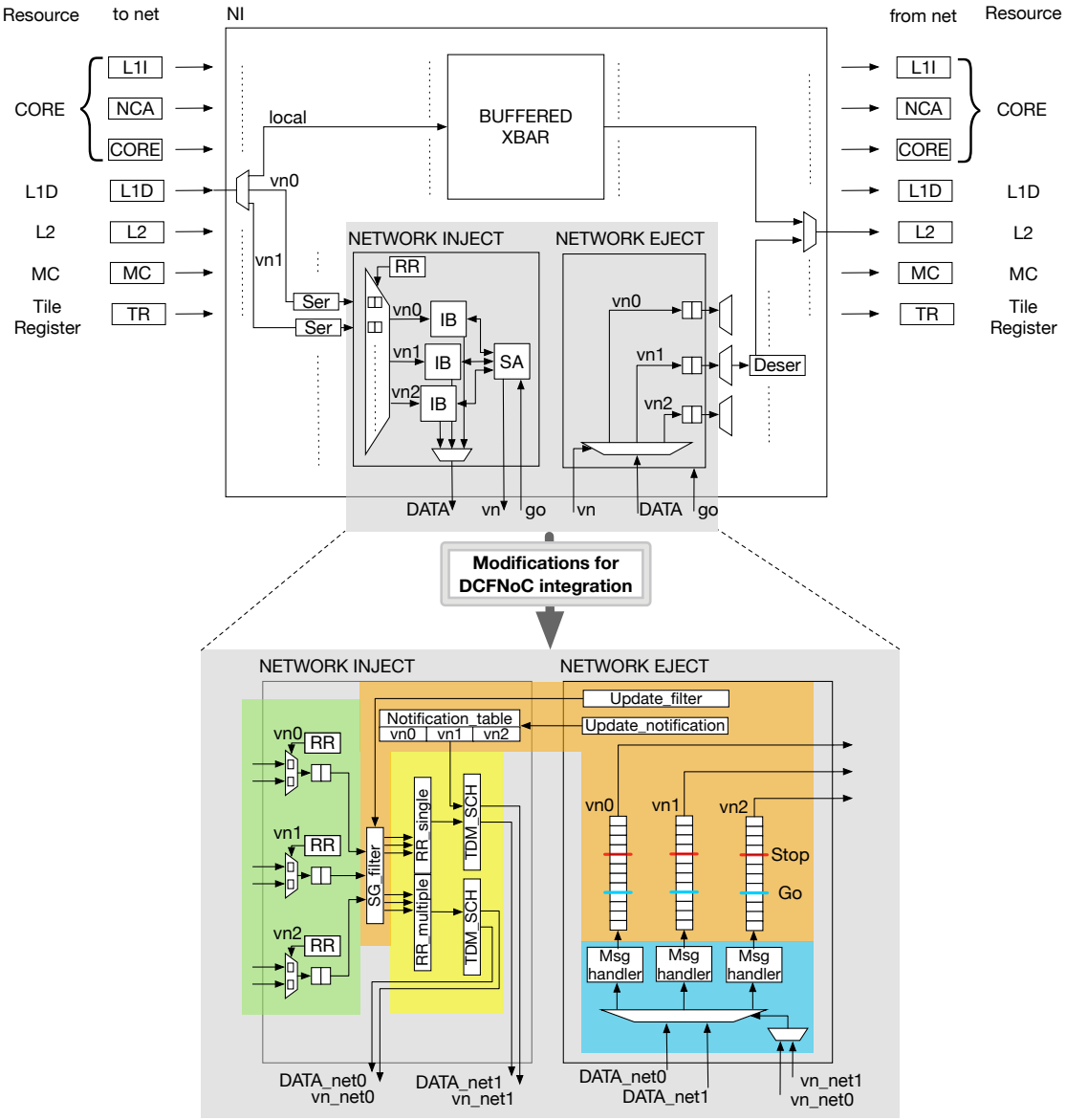


FIGURE 4.2: Network interface controller using Wormhole network and modifications to include TDM and support for two DCFNoC networks.

4.1.3 Modifications to Include TDM and DCFNoC

In order to integrate DCFNoC we modify the network inject and network eject modules. The remaining NI components are not modified. Moreover, the NoC routers will be replaced by the DCFNoC router presented above. Figure 4.2 depicts a general view of this integration at NI level. As we can see, VN multiplexing is performed at the entry point of network inject module, thereby messages corresponding to the same VN are multiplexed using a round robin arbiter and allocated at the input port queues (shaded in green). The previous large multiplexer is replaced by one multiplexer per VN.

4.1.3.1 End-to-end Flow Control

The DCFNoC routers do not implement flow control. However, end-to-end flow control will still be needed since applications may saturate end nodes. To support this functionality, the NI implements an end-to-end Stop&Go flow control protocol based on notification messages and injection filters placed at every NI module (shaded in orange). At network eject module one output buffer is allocated per VN.

When an output buffer reaches the Stop threshold the notification table is updated by using *Update_notification* signal. At the network inject module, the notification table generates a notification broadcast message to update every node filter allocated at network inject module to stop sending messages with this end node as destination. When a Stop notification broadcast is received at eject module the notification filter is updated by using *Update_filter* signal. Only outgoing messages with this destination node are blocked. Once the saturated destination node reaches the Go threshold the notification table is updated to resume the communication by sending a Go notification broadcast message. At injection time the Stop&Go filter avoids message loss when the destination node experiences saturation.

Notification messages can use preallocated slots for their transmission. Although this would impact performance (bandwidth wastage) the amount of notification messages is negligible. Only when end-point queues fill over a threshold a notification message would be sent. With a proper design, this rarely occurs. Alternatively, a side DCFNoC can be used for this light weight traffic.

4.1.3.2 Enforcing Deadlock Freedom

As DCFNoC is a buffer-less network (although it contains latches) flow control between routers is not needed. Indeed, a message that is injected into the network will be forwarded without any stop following the delays and links imposed at design time. This means there is no chance of deadlock within the network. Also, broadcast messages are injected and follow XY paths. Those messages duplicate when needed to reach all destinations. However, every message copy follows the same delay approach using the latches and therefore, can not be blocked within the network. The only deadlock that

could occur is at the edges of the network where injection and ejection buffers are used. Those buffers have been sized properly and an end-to-end flow control is used to prevent any message overflow. Deadlock is avoided by using different buffers at the edges of the network to store messages of different types (e.g. requests and responses), thus preventing protocol-induced deadlocks [30].

4.1.3.3 TDM scheduler

For TDM management a simple scheduler is implemented at the network inject module. We use a TDM scheduler with a TDM slot wheel where each slot indicates the node ID that can inject at each time slot. It is important to remark that every node has the same information stored at its TDM scheduler.

In our manycore system, every node must be able to inject a message in its assigned slot, thus, a TDM slot should be sized according to the number of flits for the largest message. However, both single-flit and multi-flit messages (six flits) have to co-exist and therefore, our scheduler needs to deal with two message sizes efficiently. Long messages are data read transactions of 576 bits long divided in 6 flits of 96 bits each. We avoid using more than six flits to achieve a short TDM period. The number of flits in a multi-flit message directly affects the TDM period length.

A trivial approach to deal with multiple message sizes is to define TDM slots of different sizes and send single-flit and multi-flit messages via the same network. In this design, the TDM slot wheel needs to combine single-flit slots and multiple-flit slots for every node ID resulting in a longer TDM period for both short and long messages. A TDM period determines the average and maximum amount of time every message is waiting to be injected into the network and also proportional to the injection bandwidth. Consequently, a longer TDM period implies lower performance guarantees.

To improve performance guarantees, we also explore a second approach in which single-flit and multiple-flit messages are split and use different TDM schedulers and DCFNoC networks. In this setup, the short message and long message schedulers implement slots of different duration (one and six in our manycore setup). By doing this, every node is able to inject either a short or a long message, even both at the same time. Thus, performance guarantees and injection bandwidth are significantly better in this latter

approach. Assuming one assigned TDM slot per node ID and flow control notification messages using a different network, the maximum time a message gets delayed at TDM scheduler until finding its slot is $(N-1)\times(\#flits)$. For a 16-core system a short message delay time is $(16-1)\times 1 = 15$ and for long messages $(16-1)\times 6 = 90$. However, when all the messages are scheduled using the same TDM scheduler and DCFNoC, the maximum amount of time that a message can be waiting to get injected is $(N-1)+(N-1)\times(\#flits)$ being $\#flits$ the number of flits in a long message. Alternatively, in the case that flow control notification messages use additional TDM slots in the same TDM scheduler as single-flit messages, a short message delay time is $2\times(N-1)$ and long messages are not affected. On the contrary, when all messages are scheduled in the same TDM scheduler, the maximum waiting time is $2\times(N-1)+(N-1)\times(\#flits)$ being $\#flits$ the number of flits in a long message.

Figure 4.2 shows the NI implementing two TDM schedulers to separate single-flit and multi-flit messages (shaded in yellow). For the implementation of the two parallel TDM schedulers, the network inject module separates messages by length and send them using the corresponding network.

4.1.3.4 Network Ejection Module

At ejection, incoming messages are first multiplexed by VN and later the message handler module is the responsible to establish the order between single-flit and multi-flit corresponding to the same VN (shaded in blue). In case two messages from the same virtual network are incoming, the message handler serializes the messages in order to preserve the order among the flits they contain and avoid corrupted messages. For example, if a multi-flit message composed of six flits has delivered three flits and at this moment it arrives a single-flit message, the message handler will buffer the single-flit message until the three remaining flits of the multi-flit message arrive.

Additionally, the NI implements receiving buffers from the end node. Those buffers will be used to store messages generated by the end node until the corresponding time slot can be used to inject the messages. Therefore, the NI decouples generation from the local node.

4.2 Evaluation Results

In this section we analyse DCFNoC performance guarantees in a manycore system and compare them with the ones provided by wormhole. Next, we provide a performance evaluation of real workloads. Finally, to analyze the feasibility of implementing the proposed NoC we provide area and maximum attainable clock frequency, and compare these results with the ones obtained with a standard wormhole router [28] [29].

4.2.1 Experimental Setup

We design DCFNoC and the whole manycore infrastructure using verilog RTL which can be synthetized for FPGAs and ASIC. We simulate the system using the Xilinx Vivado [58] RTL simulator. Thus, results presented in the chapter match the number of cycles of a potential manycore implementation.

For the experimental setup we use a 2D-mesh NoC topology to interconnect the different tiles of the previously described manycore system and a memory controller (MC). The MC is modeled using the IP provided by Xilinx to communicate with the off-chip DRAM memory. Additionally, to force the worst contention scenario the NoC can be fed by a message system generator (MS) implemented at each network interface using uniform traffic pattern. DCFNoC configuration used includes 96-bit width links to implement a mesh network topology. At network edges we statically allocate 16 1-cycle time slots to nodes. One time slot is equal to 1/16 of total bandwidth, resulting to a 6.25%.

4.2.2 Timing Guarantees

To analyze the performance guarantees of the manycore with DCFNoC, we have designed a kernel application in which we can vary the percentage of requests to the NoC with respect to the total number of instructions. To do so, we inject random non-memory operations in a kernel containing a specific number of cache accesses that miss in L1 and L2 caches. We deploy this benchmark with the ability to have three different percentages of memory accesses (2%, 7% and 15%) in order to understand the impact of the communication of a task in the performance guarantees the same can achieve.

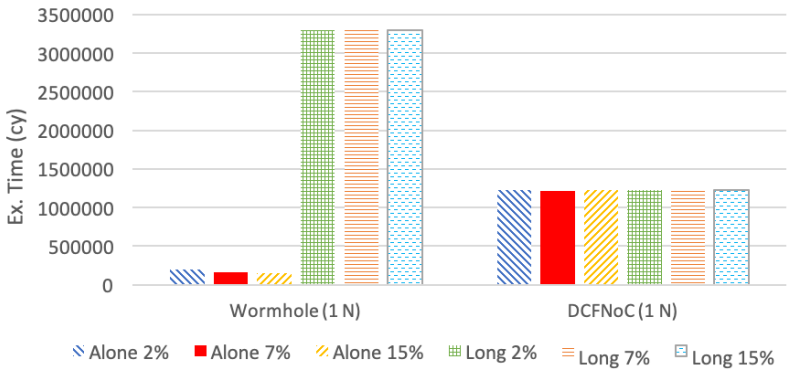


FIGURE 4.3: Execution time for benchmarks with different percentage of memory access instructions.

To generate a worst-case contention scenario in the NoC we replace the regular cores in the remaining tiles (all but tile 15) with synthetic message generators with destination router 0 (in which memory controller is placed). See Figure 4.1 that describes the NoC topology.

First, we take measurements of the benchmark running in core 15 when message generators are disconnected, to have the baseline timing measurement (*Alone*) for both DCFNoC and wormhole. Later, we switch message generators on, in all cores but the one running the task under analysis, to measure the impact of NoC interference in our application for both designs. For this experiment, message system generators inject multi-flit (*Long*) messages. Figure 4.3 shows total execution time of the different benchmark versions when they are executed alone (*Alone*) and when other nodes are injecting long messages (Long) at the maximum speed. The first observation we make is that as expected when the task is executed *Alone*, DCFNoC execution time is higher than when using the wormhole setup. The reason for this is that DCFNoC inherently restricts the injection of messages since they can be only injected in the assigned slot. Additionally, since we are only using one NoC the TDM scheduler period is high. The slowdown introduced by DCFNoC in this case is 6.21×, 7.72×, and 8.22×, for the 2%, 7% and 15% benchmarks, respectively. However, as we will show later, this slowdown is reduced when using two TDM networks. On the contrary, when the task in executed in a high contention scenario the performance of the wormhole setups is degraded significantly (around 19.85× in average) while the performance of DCFNoC is simply unaffected.

Another interesting observation is that the percentage of NoC requests does not have a

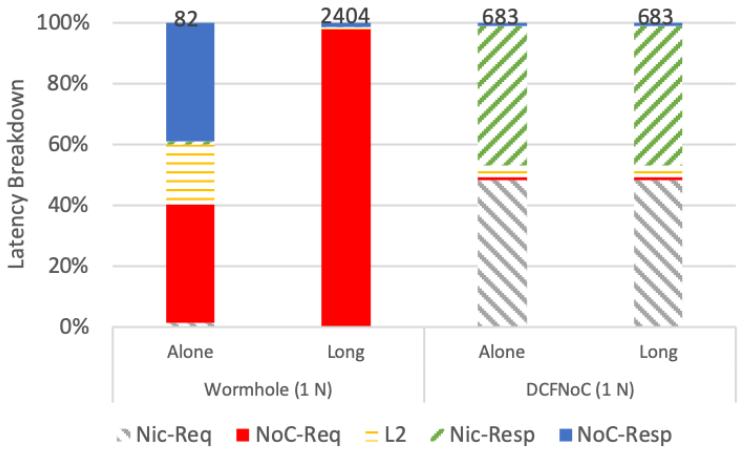


FIGURE 4.4: Average memory transaction latency when using only one network.

significant impact in the slowdown of the application in wormhole. Under such heavy message injection, the NoC gets saturated quickly. However, requests to the NoC from the task in progress keeps always at the same speed once the NoC is saturated and in all benchmarks we have the same amount of memory requests. Note also that since we are focusing in NoC contention, we avoid end-node contention by ejecting messages at 1 flit/cycle. Given that the percentage of NoC accesses does not have a significant impact in the worst-contention we use only one of the kernels in the remaining experiments.

4.2.3 Performance Guarantees of DCFNoC Manycore System

In order to characterize how NoC interference affects to applications running in the manycore system using DCFNoC, we launch the benchmark with 2% of memory instructions using only one NoC. In this manycore configuration, we can have short (1-flit) and long (6-flit) messages that correspond to NoC requests and responses, respectively. Control messages required by the coherence protocol are 1-flit long.

Figure 4.4 shows average memory transaction latency breakdown. Total average latency in cycles is shown at the top of the bars. We break down the total average latency in five components. The first two are related to request process, then memory time (L2) and later two for response part. Request and response contains NI waiting time that corresponds to waiting time for inject at the assigned time slot and later the network latency.

Wormhole baseline NoC latency is 32 cycles (4 for injection plus 5 cycles per router and ejection), while DCFNoC takes only 8 cycles (6 for the longest minimal path plus the injection and ejection).

On the contrary, when other tiles inject messages (Long scenario), wormhole NoC experiences message interference along the network and gets congested producing a huge increase of messages waiting time at input port buffers. As a consequence, the application suffers prolonged execution time and long NoC request latencies as shown in Figure 4.4 (second column). Note that wormhole NoC suffers an average latency of 2404 cycles in this case while DCFNoC average latency keeps constant to 683 cycles. Note that although DCFNoC latency is guaranteed to be equal to the diameter of the NoC the latency of the packets is higher since they are enqueued at the NI. The small TDM period of DCFNoC decreases the average time each message is waiting until it is aligned with the assigned slot and also enables DCFNoC achieve smaller latency values for higher injection rates.

In order to improve the performance guarantees for both wormhole and DCFNoC we separate short and long messages in two different networks. This allows us to reduce TDM period for short and long messages. Since the TDM period defines the average amount of time every message is waiting to be injected into the network, messages suffer less NI waiting time with this configuration. For wormhole, we also analyze the performance when splitting messages in two virtual networks (VN). The messages share physical links between routers while they use separate input port buffers. Using separate buffers in wormhole avoids head of line blocking problem [26].

As Figure 4.5 shows, we analyse the application when running alone (*Alone*), when other tiles inject short messages (*Short*) and when inject long messages (*Long*). These three scenarios are compared in a system using one wormhole network, when using two wormhole networks either virtual or physical (one for short and one for long messages), and when using only one DCFNoC for short and long as well as when using two DCFNoC networks as explained in Section 4.1.

As Figure 4.5 shows, DCFNoC improves execution time of the application by 3.7× with respect to wormhole when short and long messages are divided in two different NoCs. This is due to TDM period reduction. On the contrary, for wormhole splitting short

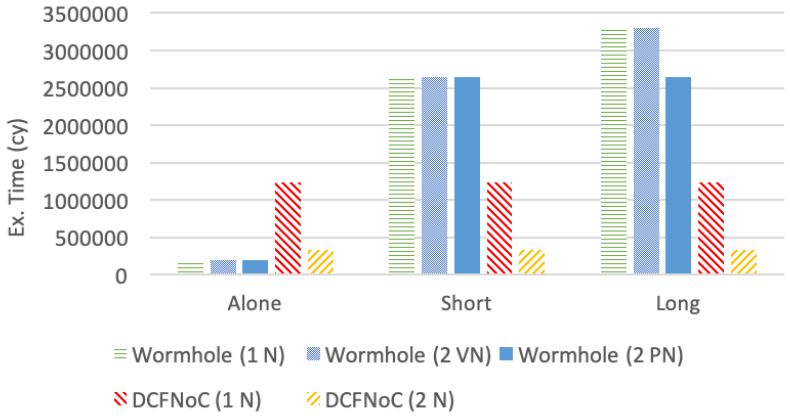


FIGURE 4.5: Execution time when using only one or two networks.

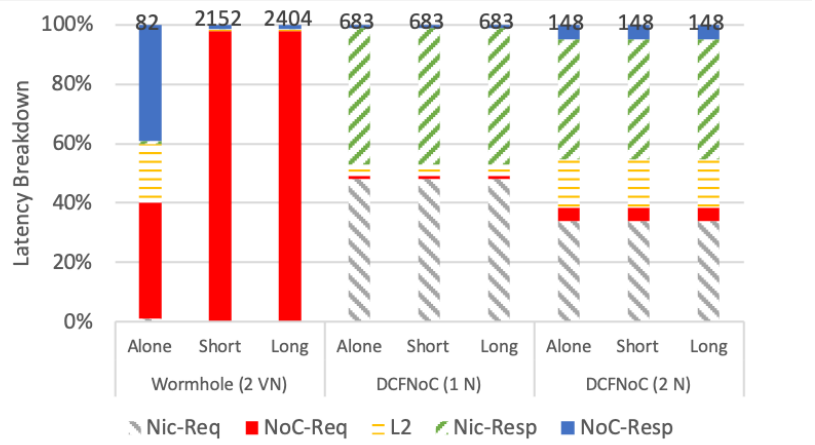


FIGURE 4.6: Average memory transaction latency when using only one or two networks.

and long messages in different NoCs does not reduce the contention suffered by our application. This is explained by the fact that wormhole does not restrict the injection of messages as DCFNoC, therefore, both wormhole networks get saturated. As the previous case, when the application runs alone in the system the wormhole network takes less execution time than DCFNoC but when using two NoCs the slowdown is only 1.65× as depicted in Figure 4.5.

Figure 4.6 shows average memory transaction latency. As shown in the figure, when the application is exposed to maximum contention the wormhole network does not guarantee performance and suffers network interferences increasing NoC request time due to long waiting time of messages at input port buffers. In contrast, DCFNoC preserves bandwidth isolation regardless the amount of network messages and network interferences.

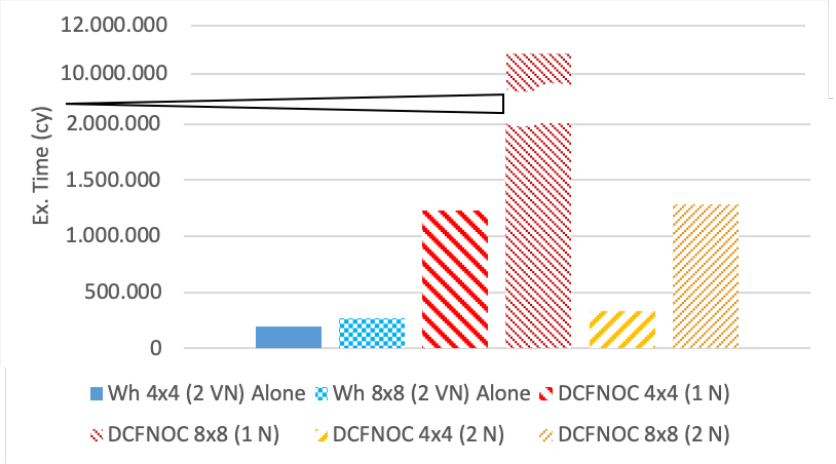


FIGURE 4.7: Scalability of execution time for 4×4 and 8×8.

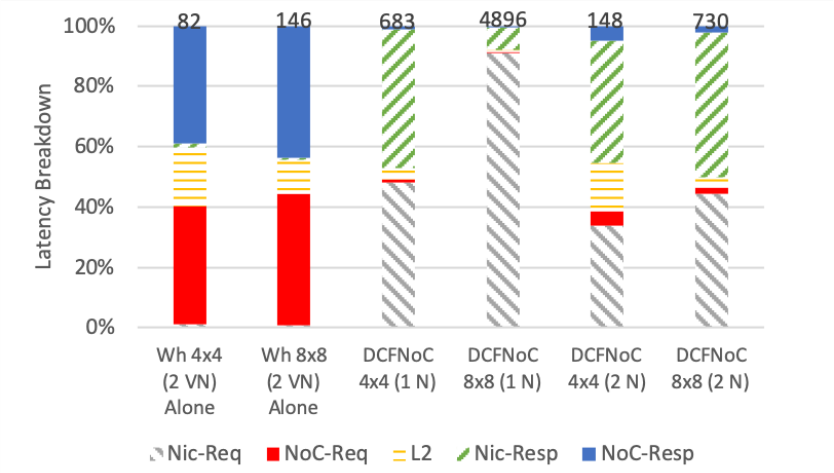


FIGURE 4.8: Scalability of average memory transaction latency for 4×4 and 8×8.

To analyse the scalability of DCFNoC, we model a 8×8 manycore system. Figure 4.7 shows execution time values for a benchmark with 2% of memory instructions when it is executed in the farthest node in the NoC. Unfortunately, we were not able to obtain execution time values for the highest contention scenario for the 8×8 setup with wormhole and thus, we only report execution times of the application executed alone for this NoC. The reason is that when our application is running in core 63 and all other cores are injecting messages at the maximum speed the bandwidth reduction experienced by wormhole is so vast that makes practically impossible to run the application in an detailed RTL simulator as the one we use.

As shown in Figure 4.7, wormhole NoC running alone has higher execution time since messages need to traverse longer paths when moving from 4×4 to 8×8 NoC sizes. When

using DCFNoC the increment of execution time is caused by the network size which in turn affects the TDM period and path length. Execution time impact when using one and two DCFNoC networks is 8.8× and 4×, respectively. Note also that although DCFNoC performance decreases, meshes of this size usually have more than one memory controller which reduces the longest paths and allows improving DCFNoC performance.

Figure 4.8 shows average memory transaction latency. As shown in the figure, request waiting time increases when moving to a 8×8 mesh by 7.17× and 4.93× for one and two DCFNoCs, respectively. However, NoC latency is only affected by hop count being 8 cycles in 4×4 and 16 cycles in 8×8.

DCFNoC provides performance guarantees that are superior to the ones wormhole provides even for smaller NoCs. DCFNoC guaranteed performance for a 8×8 NoC are 2.06× better than those obtained for a 4×4 wormhole NoC. Although wormhole NoC performance is higher when implemented in COTS manycores, it introduces a significant negative impact in the quality of WCET estimation, preventing any assumption on affordable timing quality of messages. Contrarily, DCFNoC provides perfect timing isolation as well as constant network message latency to fulfill safety-related applications requirements in manycore systems.

4.2.4 Flexible Bandwidth Allocation in the Manycore

In order to test the capabilities of the flexible bandwidth allocation of DCFNoC, we perform an experiment on a 4×4 mesh using uniform traffic. Figure 4.9 depicts application execution time when running alone (*Alone*) using wormhole NoC, also when other tiles inject long messages using DCFNoC with 1/16 of total bandwidth, moreover with a bandwidth of 2/17.

As figure shows, application execution time using DCFNoC with a bandwidth of 1/16 increases by 2.14 times compared with *Alone* scenario when using wormhole NoC. As expected, when the application node has 2-cycle assigned time slots of 17, execution takes only 1.6 times. Note also the fact that as Figure 4.10 shows, transaction latency is improved by 35% due to important NI average waiting time reduction.

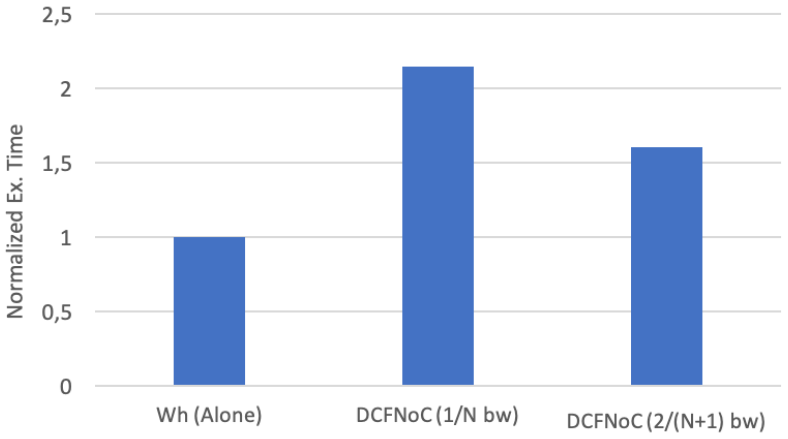


FIGURE 4.9: Execution time using different bandwidth allocations. Application executed alone using a wormhole NoC and using DCFNoC with 1-cycle assigned time slot of 16 and 2-cycle assigned time slots of 17.

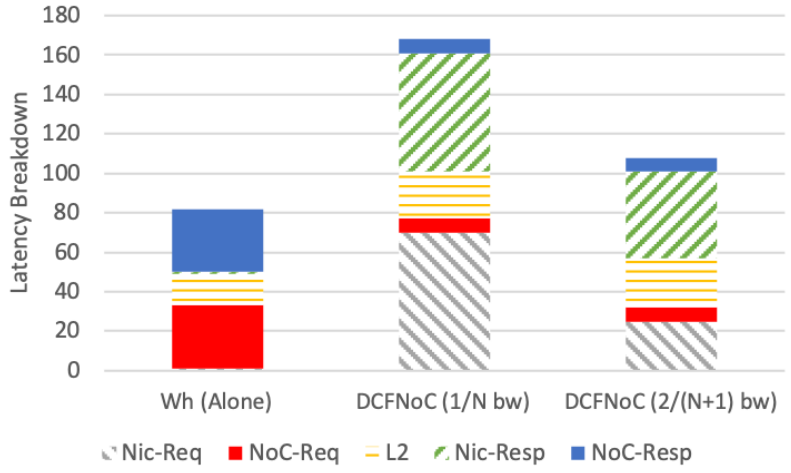


FIGURE 4.10: Average transaction latency using different bandwidth allocations.

4.2.5 Performance Evaluation of Real Workloads in a Manycore

In order to evaluate the benefits of using DCFNoC in safety-related applications, we have selected (*ndes*) and (*matmult*) benchmarks from mälardalen WCET benchmarks suite [59]. Applications in this benchmark suite have small memory footprint and thus, have very low communication requirements. For comparison purposes we also include a kernel application (*synth*) resembling applications with higher communication needs (5% of total instructions performing NoC requests). Note that large applications cannot be effectively simulated in a detailed RTL manycore model.

The three applications are evaluated in two scenarios: low and high contention. To create

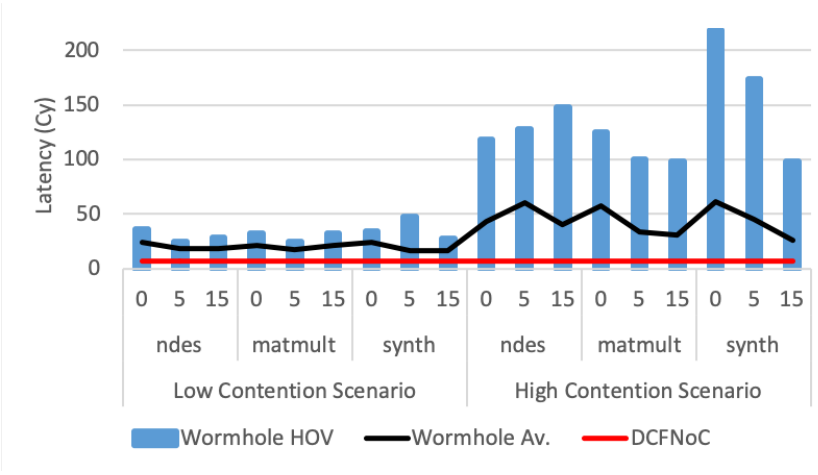


FIGURE 4.11: Latency for different benchmarks execution at different core positions $(0, 5, 15)$ in a 4×4 mesh system using wormhole and DCFNoC NoCs.

this contention scenarios we use synthetic message generators with random destinations injecting long messages at a 6.66% and 100% injection rates. We perform experiments placing the kernels at three different cores $(0, 5, 15)$ in a 4×4 system.

Figure 4.11 shows a latency comparison for these three benchmarks running in cores $(0, 5, 15)$ in low and high contention scenarios when using wormhole and DCFNoC NoCs. For wormhole, we present Highest Observed Values (HOV), represented by blue bars, and average values, represented by a black line. As shown in the plot, the largest HOV is obtained for the kernel with higher communication needs (synth) and in the high contention scenario. This is explained by the fact that high contention conditions are more likely to occur when the NoC is congested. Unfortunately, for hard-real time systems it is not possible to assume HOV represents actual contention bounds. In fact, as shown in [66] the worst possible contention is possible with few NoC requests if they aligned in the worst possible manner. Finally, DCFNoC latency keeps always constant being its value much lower than the average and HOV values of wormhole.

4.2.6 Area and Frequency of DCFNoC

Maximum operating frequency and area utilization are obtained using Cadence RC Compiler and the 45-nm Nangate library [60]. The wormhole NoC implemented is 64-bit width and uses 8 slot input buffers with Stop&Go flow control. For the implementation results we consider a DCFNoC router for a 8×8 mesh. The wormhole router (WH) used for

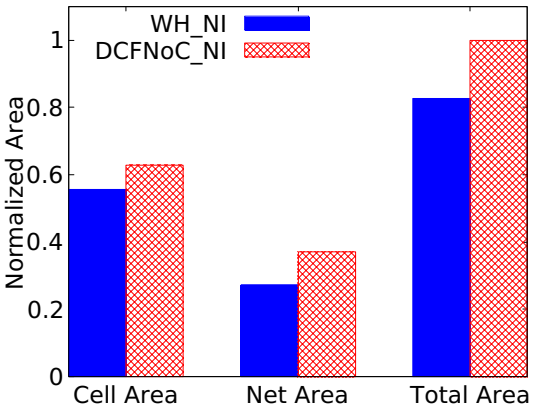


FIGURE 4.12: Network Interface Area overhead.

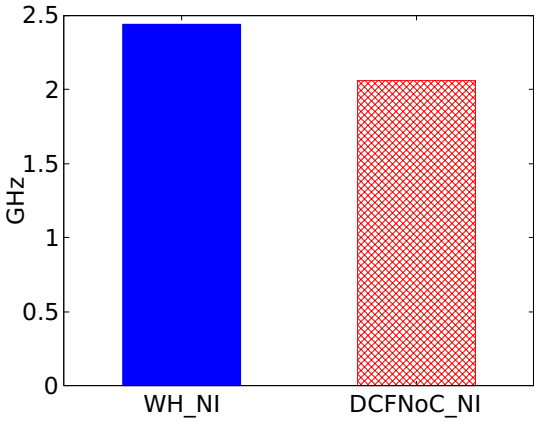


FIGURE 4.13: Network Interface Maximum attainable clock frequency.

comparison purposes implements a single virtual channel, virtual networks are not used in this particular router.

Figure 4.12 shows area overheads for the two NI when targeting high frequency. The DCFNoC NI uses 7.3% more cells than the one used for wormhole and a total area of $25,750\ mm^2$. This results in an increment of 17.3% of total area with a total area of $31,137\ mm^2$. Even though wormhole NI multiplexing logic is more complex, it does not require an end to end flow control and therefore consumes less area. The DCFNoC NI uses logic to implement VN multiplexers, TDM arbiters and flow control buffers. End-to-end flow control uses one output buffer per VN.

As a result, total area overhead of DCFNoC router and NI is $60,072\ mm^2$ and $67,336\ mm^2$ for wormhole. In summary, we obtain total area savings of 10.79%.

We also analyse the NI maximum attainable frequency. Figure 4.13 shows a slowdown of 15.57% in clock frequency of DCFNoC NI w.r.t wormhole. Although wormhole NI is more complex, it is pipelined in several cycles which allows achieving higher frequencies. On the contrary, TDM arbiters and flow control of DCFNoC NI use only one cycle simplifying flow control notifications. Even with such critical path, DCFNoC NI clock frequency reaches up to 2.06 GHz. Other flow control schemes can be considered in the future to allow reaching higher clock frequencies.

4.3 Summary

In this chapter, we present a manycore system integrating a novel real-time NoC (DCFNoC) to enforce predictability. We show that DCFNoC can be smoothly integrated into a manycore design by introducing small modifications at network interface. Our results confirm that the resulting manycore provides performance guarantees that are significantly better that the ones that can be achieved with wormhole NoC designs.

In the next chapter we target performance improvements of DCFNoC in non congested/saturated conditions.

Chapter 5

hp-DCFNoC: High Performance Distributed Dynamic TDM Scheduler based on DCFNoC Theory

In a context in which NoCs are becoming ubiquitous in safety-critical real-time systems [67][38][63] it becomes mandatory finding NoC designs that provide high quality real-time guarantees. In this chapter, we aim at achieving this goal and propose a new real-time specific NoC design that provides peak performance close to the standard wormhole designs while preserving strict real-time guarantees. To achieve this, our NoC design uses a dynamic scheduler that builds on top of DCFNoC. The combination of the dynamic scheduler and DCFNoC is termed hp-DCFNoC (high-performance DCFNoC).

We implement the scheduler design in synthesizable verilog RTL showing the feasibility of this approach and compare its performance against the one provided by DCFNoC and regular wormhole NoC.

This chapter [1] is organized as follows. First, it presents the distributed dynamic scheduler design in Section 5.1. Next, a rescheduling technique to improve scheduler performance is described in Section 5.1.5. Later, performance evaluation of the proposed scheduler is

[1] The work discussed in this chapter has been published in [68].

provided in Section 5.2.3. Finally, area overhead and maximum attainable frequency are analysed in Section 5.2.5.

5.1 A Distributed Dynamic Scheduler Design

DCFNoC limits flit injection to only one node at a time in a given slot. Therefore, although timing guarantees are preserved, network throughput is severely limited to one flit per cycle regardless of network size. hp-DCFNoC overcomes this limitation by introducing a dynamic scheduler design that is able to inject more than one flit per cycle by exploiting the use of conflict-free paths (paths that do not share any network resource between them). In particular, we exploit two conflict-free situations: (1) packets from two nodes injected in the same slot that do not share any resource along their path and (2) messages injected at different cycles. In both cases they will never conflict in DCFNoC. Our dynamic scheduler exploits these two properties to maximize the number of packets that can be injected in the NoC at a given cycle while preserving the real-time properties provided by DCFNoC.

hp-DCFNoC implements a distributed TDM scheduler at network interfaces of every end node. The scheduler uses two DCFNoC networks, one for notification and one for data transmission as shown in Figure 5.1. Two phases are identified. In the notification phase the scheduler determines which routes will be used by each node in a given time slot. If routes are compatible (i.e. do not generate conflicts) they can be scheduled in the same time slot. In the second phase data is actually transmitted. During the transmission phase all the slots are run. Data transmission and notification phases are overlapped to maximize both throughput and average message latency.

5.1.1 Scheduler Architecture

The proposed scheduler consists of several modules interconnected as shown in Figure 5.2. This picture shows a detailed architectural diagram of a 4-way distributed TDM scheduler for a 16-node system in a 4×4 mesh network. Each way_i module has a queue with pending message, the node wants to inject (4 messages in this particular configuration). In the scheduler, a route scheduling table (RST), with as many entries as ways keeps

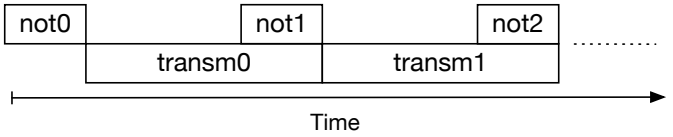


FIGURE 5.1: Scheduler phases. Notification phase configures next transmission window.

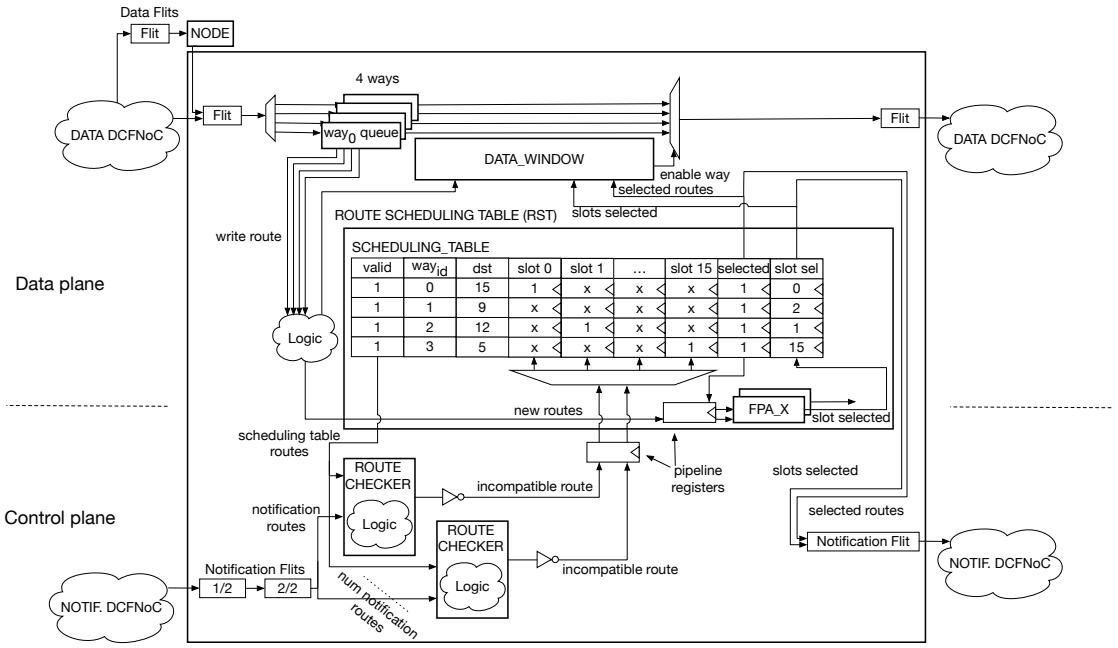


FIGURE 5.2: Scheduler to optimize DCFNoC performance and resource utilization.

the following information related to pending messages: valid bit (*valid*), way id (way_{id}), destination node (*dst*), the set of available slots (*slot x*), a selected slot vector (*selected*) and the corresponding id of the selected slot (*slot sel*).

During the scheduling phase, each node receives notification messages from other nodes via the notification DCFNoC. Every time a notification arrives to the scheduler the route checker modules compare the routes information included in the received notification with the one for the pending local messages. A notification message can include several routes. In our implementation we use slots of two cycles for each node to send notifications. Each cycle $ways/2$ routes are notified. By using two cycles the width of the notification network is reduced. The route checker modules determine which routes are disjoint with the pending messages and which ones are incompatible. The selected slot is updated in the RST module using a chain of Fixed Priority Arbiters (FPA). Once notification phase ends, RST information is stored in the data window module. This module injects pending

messages in the assigned slot by using an enable signal during the data transmission phase.

In order to maximize performance, the scheduler is pipelined in two stages. In the first stage, notifications are received (or injected) and checked. In the second stage, the compatible slots are found.

5.1.2 Notification Phase

At each network interface, newly generated messages arrive in order, and are stored in a two slot buffer (way). Each message generates a pending route to be scheduled (the suitable slot must be computed). Routes are determined by a source and destination pair (src-dst). Message destination and way ID are stored in the route scheduling table (RST), which is used to find the most suitable slot. Each RST row contains one control bit per each possible slot in the next transmission window. A control bit set to one means that this route can use that slot ID for transmission. The scheduler must guarantee two conflicting paths do not end up using the same slot. We define as many slots as end nodes and statically assign one slot per node. Each slot will be prioritized by the scheduler to the assigned node. Thus, the scheduler guarantees that at least one node will be able to use its prioritized slot and is irrevocable. This is the most valuable guarantee, no one can use this slot unless it uses a disjoint route. This is key to ensure that DCFNoC timing guarantees are preserved.

The notification phase uses a TDM network in order to let every node send their notifications in turns. To avoid wasting a complete TDM window to notify all nodes, we use DCFNoC native support for broadcast. Thus, every node will broadcast its notifications on a single slot to the rest of nodes. Also, DCFNoC guarantees all notifications arrive at the same time to all nodes which simplifies the scheduler design. The notification reception time ($time_i$) identifies the sender node and therefore is equal to $Notification\ ID$. On every notification window, the first node sending notifications is assigned in a round-robin fashion.

On every notification reception the end nodes process the notifications. The RST is updated taking into account the conflicts that may arise between received notifications and the current assignments of paths to slots. Notice that all nodes update the RST

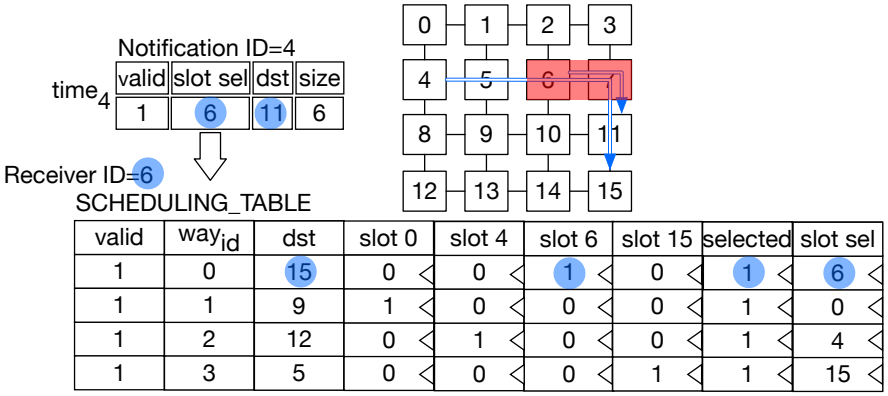


FIGURE 5.3: First priority rule in case of incompatible paths: Notified route uses the priority slot of the receiver.

at the same time and via the same manner. Also, each end node checks if there is a pending route (i.e. a route that is waiting to be served) that requests the same network resources to determine if these routes are compatible. The rules used by the scheduler to determine priorities in case of incompatible paths are the following:

- The notified route uses the priority slot of the receiver. For the examples provided below we consider TDM slot i is assigned to node i by default. Given that these routes are compatible by default no action is required at the receiver side. Figure 5.3 illustrates this case. Notification node 4 uses priority slot of receiver (slot 6). Receiver has one route in the RST with slot 6. In this case, the receiver has the priority to use this slot. When the notification turn arrives, the receiver node will send the notification and other nodes using this slot with incompatible routes must disable this selected time slot in the RST module.

- The notified route uses the priority slot of the sender. The receiver must disable the requested time slot in the RST. Figure 5.4 shows an example where notification node 4 uses its own priority slot (slot 4). Receiver (node 6) has one route in the RST with slot 4 and both routes have incompatible paths. In this case, the sender has the priority to use this slot and therefore, receiver node must disable this selected time slot in the RST row.

- The notified route does not use either the priority slot of the sender or the receiver's one. Figure 5.5 illustrates this case. The sender is node 4 and uses slot 15. The receiver is node 6 and has one route in the RST with slot 15. Both routes have incompatible paths due to destination node sharing. In this case, the first node

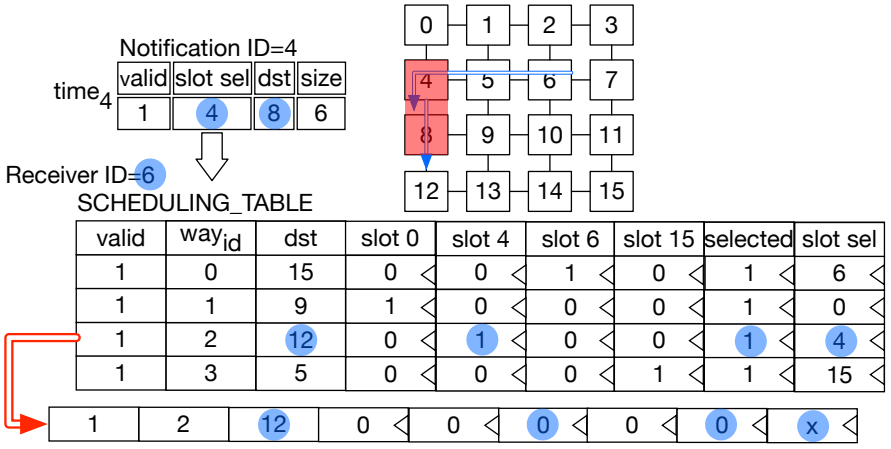


FIGURE 5.4: Second priority rule in case of incompatible paths: Notified route uses the priority slot of the sender.

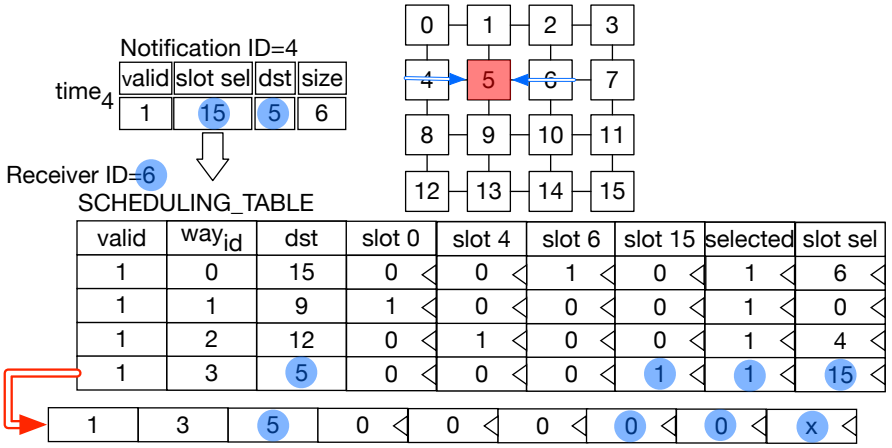
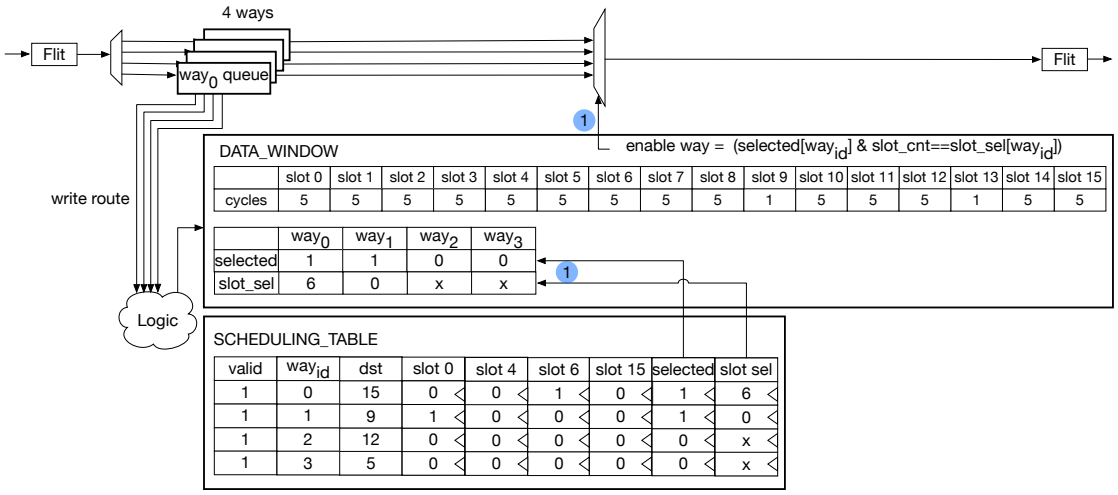


FIGURE 5.5: Third priority rule in case of incompatible paths: Notified route does not use either the priority slot of the sender nor the receiver's one.

that notified the route (node 4 in our example), has the priority to use this slot, therefore, the receiver node must disable this selected time slot in the RST row.

Once notification turn arrives, the slot manager on every node selects one data slot for each pending route through a chain of FPA. Slot selection process is based on the updated control bits of the RST. First, by means of a Round-Robin arbiter, one pending route is selected and uses the first FPA to select a time slot. The FPA selects the first active bit of each RST row starting from this node's priority slot. The remaining pending routes use the next entry of the one provided by FPA. It is important to take into account that the slot selection process is exclusive. That is, no pending route can select the same slot. Once slot selection ends the selected control bit is enabled and the selected slot is stored at the row.



	slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7	slot 8	slot 9	slot 10	slot 11	slot 12	slot 13	slot 14	slot 15
cycles	5	5	5	5	5	5	5	5	5	1	5	5	5	1	5	5

	way_0	way_1	way_2	way_3
selected	1	1	0	0
slot_sel	6	0	x	x

valid	way_{id}	dst	slot 0	slot 4	slot 6	slot 15	selected	slot sel
1	0	15	0	0	1	0	1	6
1	1	9	1	0	0	0	1	0
1	2	12	0	0	0	0	0	x
1	3	5	0	0	0	0	0	x

FIGURE 5.6: Data phase. At the end of notification phase the slot manager sends information about the slots assigned to each message.

Selection and notification process take two cycles at each node. The process selects and notifies half of the pending routes in the first cycle and the other half in the second cycle.

5.1.3 Data Phase

After the notification phase each node knows the exact slot in which the different messages have to be scheduled. The duration of the data phase that guarantees all nodes have one slot is a complete TDM window. The size of each slot is the maximum size among all the routes using the slot and depends on the size of messages scheduled for that slot.

To maximize performance, transmission phase is overlapped with notification phase. As Figure 5.6 shows, when a notification phase ends, the slot manager module sends information about the slots assigned to each message. Once the new data window is ready the system starts sending stored messages from ways corresponding to an assigned slot.

The upper part of Figure 5.8 shows how the data window module activates the notification phase of the next data transmission phase before the current transmission window is completed. Note also that shifting notification phase to the end of the current data window transmission maximizes the chances to find compatible routes, since more messages are potentially available.

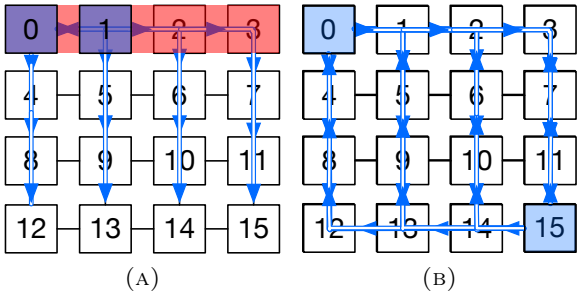


FIGURE 5.7: Disjoint paths between node 0 and node 1 in (a). In (b) node 0 and node 15. Paths sharing red resources or destination node are incompatible.

5.1.4 Assignment of TDM slot priorities

The success rate of slot assignments for pending routes is significantly influenced by how priority slots are designated to the different nodes. Achieving a higher rate of selected slots requires finding the maximum number of disjoint paths (i.e. without conflict) and for this a policy for slot assignation has to be designed. In our design, this assignment is computed by software before the application is deployed.

Let us illustrate this with an example. Figure 5.7a shows a 4x4 network in which packets are routed using XY. As shown in the plot, nodes 0 and 1 have many potential conflicting paths since they share many links to the potential target destinations. On the contrary, as shown in Figure 5.7b, node 0 and node 15 do not have conflicts except those that target one of these two nodes as destination. For this setup, the best slot assignment is the one that assigns priority slots to nodes with no or few number of sharing resources for all potential target destinations.

In this section, we show how to find the optimal slot assignment for uniform traffic with XY routing in a 2D mesh network. To do so, we have performed an exhaustive search with an offline program to get the configuration with less number of conflicting paths. The pseudo-code of this search is shown in Algorithm 5.1. First, in lines 13-16 the priority slot is copied to the *temp_map* structure. Then, for every way in the scheduler (lines 18-24) the selected slot is assigned based on the priority slot, the way id, and the number of TDM slots. Notice that all the possible destination nodes are also considered to further check the disjoint routes (lines 26-31). This function, checks the number of disjoint paths for this *temp_map* configuration. To do so, for all potential paths and the number of ways, takes into account the selected slot, and saves the best configuration

```
1: function build_pslot_map(#ways, #tiles, #slots)
2:
3:   tiles temp_map
4:   tiles best_mapping
5:   tile tx
6:   tile ty
7:   way  wx
8:   int temp_disjoint = 0;
9:  int best_disjoint = 0;
10:  destination dx
11:
12:  for every tile in tiles (tx)
13:    temp_map(tx).prio_slot = get_pslot(slots);
14:    for every tile in tiles (ty)
15:      temp_map(ty).prio_slot = get_pslot(slots);
16:    endfor
17:
18:    for every way in ways (wx)
19:      temp_map(tx).way(wx).sel_slot =
20:      (temp_map(tx).prio_slot + wx) % slots;
21:      for every tile in tiles (dx)
22:        temp_map(tx).way(wx).dest = dx;
23:      endfor
24:    endfor
25:
26:    temp_disjoint = check_disjoint(temp_map);
27:    if(temp_disjoint>best_disjoint)
28:      best_mapping = temp_map;
29:      best_disjoint = temp_disjoint;
30:    endif
31:  endfor
32:
33:  return best_mapping;
34:
35: end function
```

Alg. 5.1: Algorithm to get best priority slot mapping

in terms of number of disjoint routes. In other words, the algorithm explores for all potential paths (i.e. source and destination pair) in the network, which is the priority assignment that maximizes the number of disjoint routes. Note that the algorithm also takes into account the number of ways in the scheduler.

5.1.5 Rescheduling technique

As explained in Section 5.1.1 the proposed scheduler overlaps notification and transmission phases (see the upper part of Figure 5.8). However, as notification and data transmission phases have large timing differences, performance may be compromised. To solve this drawback, we propose a rescheduling mechanism that finds the best way to seize notification and timing of each phase.

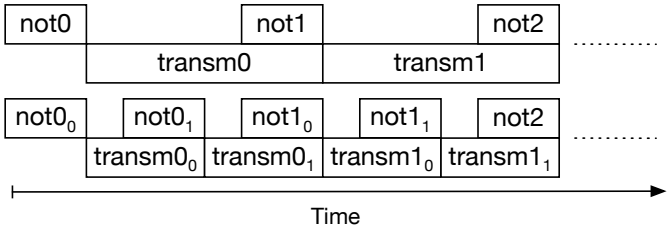


FIGURE 5.8: Comparison between common scheduler phases (up) and using rescheduling technique (down). Notification phase configures next transmission window. Data transmission phase is now broken down in $transmX_0$ and $transmX_1$.

Let us illustrate the proposed mechanism with an example. The timing of the notification phase depends on the number of nodes ($\#nodes$), the number of cycles required for notification ($\#noti_{cy}$) for each node and the network latency (net_L).

$$Notification\ delay = (\#nodes \times \#noti_{cy}) + net_L \tag{5.1}$$

Data delay depends on the number of nodes ($\#nodes$) and the message length in cycles ($message_L$). Both Notification and Data delay refers to the maximum possible delay.

$$Data\ delay = \#nodes \times \#message_L \tag{5.2}$$

A 4×4 DCFNoC mesh has a latency of 7 cycles and needs 2 cycles for node notification. Using messages of 5 flits:

$$\begin{aligned} Notification\ delay &= (16 \times 2) + 7 = 39\ cycles \\ Data\ delay &= 16 \times 5 = 80\ cycles \end{aligned} \tag{5.3}$$

As we can see, there is a huge difference between the 39 cycles of the notification phase and the 80 cycles required for transmitting data. Thus, we propose a rescheduling approach in which data phase is split. The first half of window slots in a first round and the other half of the slots in a second one. After this modification notification and data delay are as follows:

$$\begin{aligned} Notification\ delay &= (16 \times 2) + 7 = 39\ cycles \\ Data\ delay &= \frac{16}{2} \times 5 = 40\ cycles \end{aligned} \tag{5.4}$$

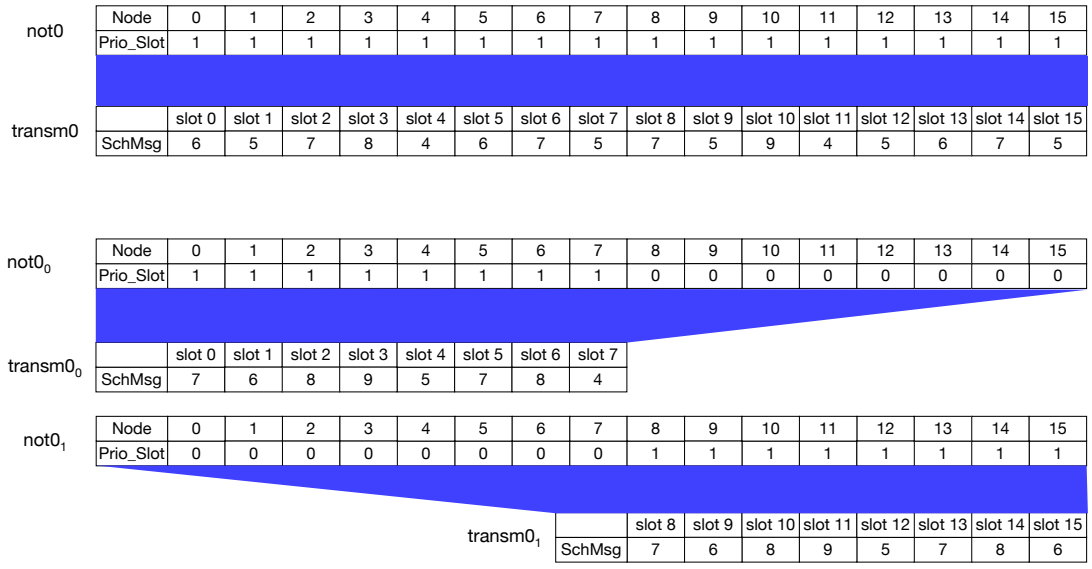


FIGURE 5.9: Example of notification nodes targeting scheduling slots in common scheduler phases (up) and using rescheduling technique (down). Note that resulting transmission phase when rescheduling is the addition of $transmX_0$ and $transmX_1$.

Notification delay remains in 39 cycles since we notify routes from all network nodes but data delay changes from 80 to 40 which results in an almost perfect overlapping.

At the lower part of Figure 5.8 we show the new procedure for notification and transmission using this rescheduling technique. Data transmission phase is broken down in $transmX_0$ and $transmX_1$. The first notification $notX_0$ deals with the first half of the data window slots and the second one $notX_1$ schedules the remaining slots.

To maximize efficiency of the rescheduling technique it is important to match the notification phase of all nodes $notX_0$ with the same or less slots for data transmission in order to maximize the matching at the end of whole data window (see the lower part of Figure 5.8).

Figure 5.9 shows how the scheduling window is organized in the regular case (top) and when the rescheduling technique is applied (bottom). In both cases, each node is assigned the priority in a given slot but routes are notified differently. In the regular scheduling, the notification phase occurs once every N slots while with the rescheduling technique the notification phase occurs several times per window (two times in this example). Notifying routes more frequently increases the chances to schedule packets. Note that nodes can use any of the slots. However, packet transmission for non-priority nodes only occur if the node with priority is not using the assigned slot.

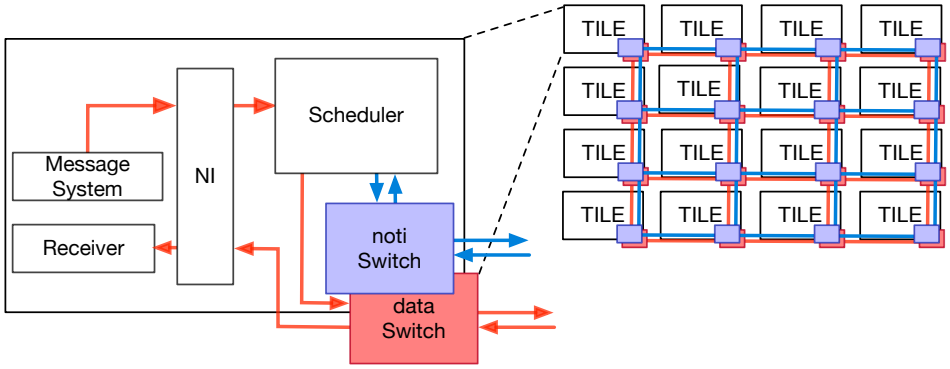


FIGURE 5.10: 4×4 mesh system with schedulers using two DCFNoC networks.

5.2 Evaluation

In this section we compare the performance achieved by hp-DCFNoC with DCFNoC and a regular wormhole NoC.

5.2.1 Experimental Setup

We implement all designs: hp-DCFNoC, DCFNoC, and a wormhole NoC in verilog RTL. The resulting implementation can be synthetized in FPGAs and ASIC. To obtain performance numbers we simulate the system using the Xilinx Vivado [58] RTL simulator.

Figure 5.10 shows a schematic of the NoC platform for the 4×4 mesh system. Note that our approach includes the schedulers and two DCFNoC networks, one for notification and one for data transmission.

For traffic generation, we create uniform traffic patterns using a message system generator that is attached to each network interface. In order to create uniform traffic pattern, we use a pseudo random number generator with Linear Feedback Shift Registers (LFSR [69]) to generate a random destination label for every message, thus all nodes have the same probability to receive a message. The use of uniform traffic allows us to simulate an unpredictable network load as well as unpredictable used paths. One thousand warm-up messages are generated at the beginning of each test. This represents a 5% of total test time.

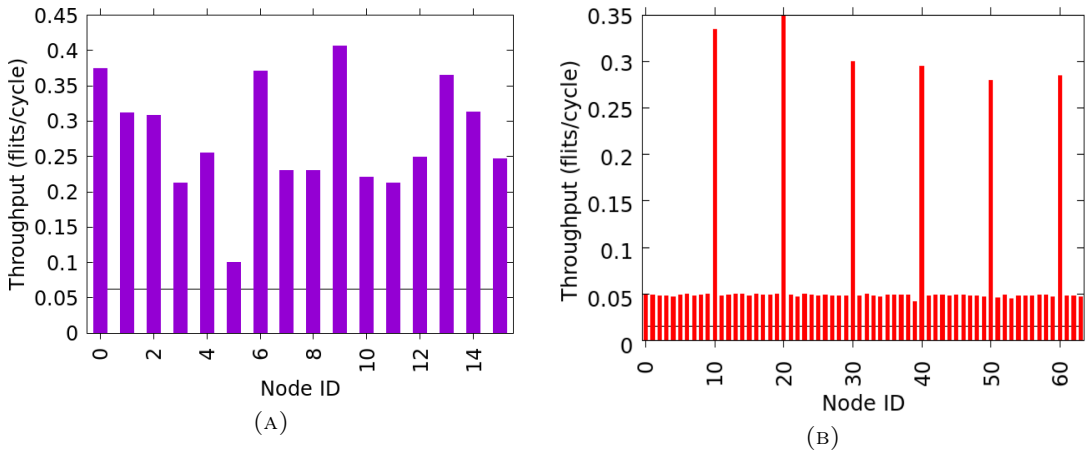
We implement NoCs of 16 (4×4 mesh) and 64 nodes (8×8 mesh) using this platform.

5.2.2 Theoretical Worst-Case Performance

hp-DCFNoC preserves the performance guarantees provided by DCFNoC network. hp-DCFNoC performance is tightly coupled with the scheduling period. While traditional TDM approaches have difficulties to find schedules for large networks, hp-DCFNoC is able to find conflict-free scenarios in arbitrarily large NoC sizes. hp-DCFNoC achieves this without degrading the quality of the achieved guarantees by simply ensuring that no more than one node is injecting packets in the same time slot unless it uses a disjoint route. Since only one node is injecting in a particular cycle the resulting NoC guaranteed productivity is equal to $1/N$ flits/cycles/node being N the number of network nodes. Hence, the network injects $1/t_{\text{slot}}$ (one message per TDM slot), at a minimum, resulting in N/P_{TDM} (N messages per TDM period), or in other words N messages per window. Regarding message latency, all communication flows experience a latency that is equal to the time required to traverse the NoC diameter (H) plus the ejection and injection links (2).

hp-DCFNoC provides TDM periods significantly better than other proposals. For instance, for a 25-node mesh NoC the approach in [1] requires a period of 34 while hp-DCFNoC requires only 25 cycles. Additionally, approaches using ILP to find optimal scheduling periods suffer from limited scalability not being able to find schedules for meshes beyond 25 nodes [1]. Other approaches based on heuristics, although being able to find schedules for larger NoCs, result in TDM periods that are significantly worse than the ones achieved by hp-DCFNoC. For instance, in [47] a 64-node mesh requires a period of 481 cycles while hp-DCFNoC requires only 64.

In terms of latency, for very small injection rates and the smallest NoC sizes DCFNoC latency is slightly worse than the one achieved in ILP [1] for the shortest paths but better for the longest ones. For higher NoC sizes and/or higher injection rates DCFNoC achieves always better results [63]. The reason for this is the smaller period of DCFNoC that decreases the average time each message is waiting until it is aligned with the assigned slot. The smaller period also enables DCFNoC achieve small latency values for higher injection rates. Other similar approaches like PhaseNoC [2] achieve the same schedule period but at the cost of higher latency. In summary, hp-DCFNoC baseline performance is in general better than the rest of state-of-the-art TDM approaches.



(A) (B)

FIGURE 5.11: hp-DCFNoC throughput guarantees for a 4×4 and 8×8 mesh system. In (a) node 5 is injecting traffic at 10% injection rate while others are injecting at 50%. In (b) nodes $(10, 20, 30, 40, 50$ and $60)$ inject at 50% injection rate while the rest inject at 5%.

5.2.3 Testing Worst-Case Performance

Figure 5.11a shows hp-DCFNoC throughput guarantees for a 4×4 mesh system. Horizontal line represents the throughput guaranteed for each node ($1/16 = 0.0625$ flits/cycle/node). For this experiment we configure node 5 to inject traffic at 10% injection rate while the rest of nodes are injecting at 50%. Note that while the guaranteed throughput is lower than the one injected by node 5 the scheduler is able to meet latency guarantees also when the other messages are injecting at a high rate. For node 5 throughput reaches 0.10 flits per cycle, above the minimum guaranteed throughput, hence the average message latency is kept above the maximum guaranteed.

Figure 5.11b analyzes for an 8×8 NoC a different traffic scenario. In this case, six nodes $(10, 20, 30, 40, 50$ and $60)$ inject at 50% injection rate while others injects at 5%. For a 64-node configuration hp-DCFNoC guarantees a throughput of 1/64 flits/cycle/node. However, with the dynamic scheduler we have that nodes injecting at 5% are able to sustain this throughput that is above the one actually guaranteed in spite of having several nodes with a very high injection rate. Note however that for nodes injecting at 50% throughput is not preserved.

Impact of the number of ways. Figure 5.12 shows the network throughput achieved for a 4×4 mesh system with the baseline scheduler (*BaseSched*) without the re-scheduling technique. As shown, throughput achieved with the scheduler is significantly better than

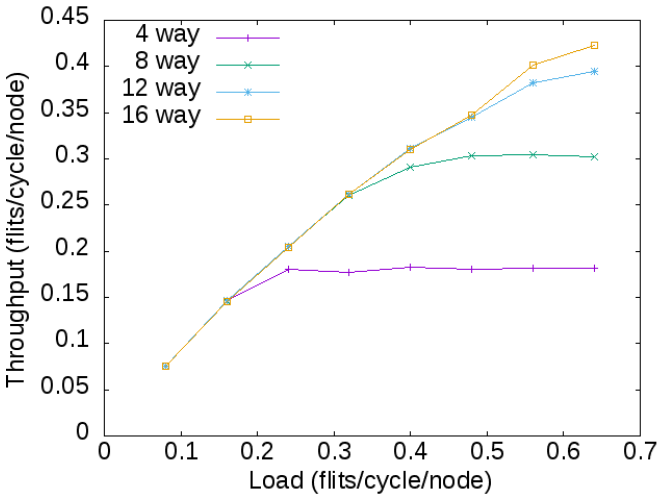
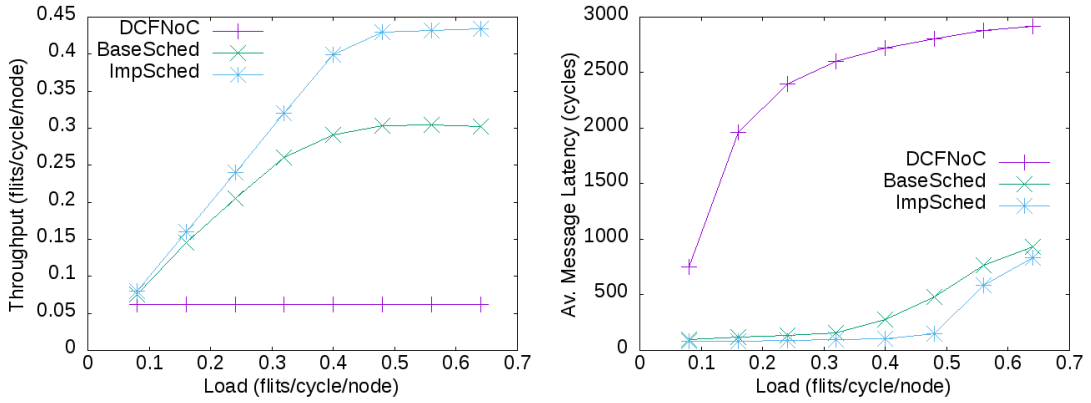


FIGURE 5.12: Network throughput using baseline scheduler (*BaseSched*) for different ways in a 4×4 mesh system.

the one achieved by DCFNoC (0.00625 flits/cycle/node) even with the a small number of ways (4). Recall that the number of ways represents the maximum amount of messages that are pending to be scheduled in each notification phase. Thus, increasing the number of ways increases the potential throughput of the network although this also implies higher hardware costs. For a 16-node NoC the throughput improves from 0.18 to 0.42 flits/cycles/node when moving from 4 to 16 ways. An interesting observation is that improvements are not so relevant for more than 8 ways.

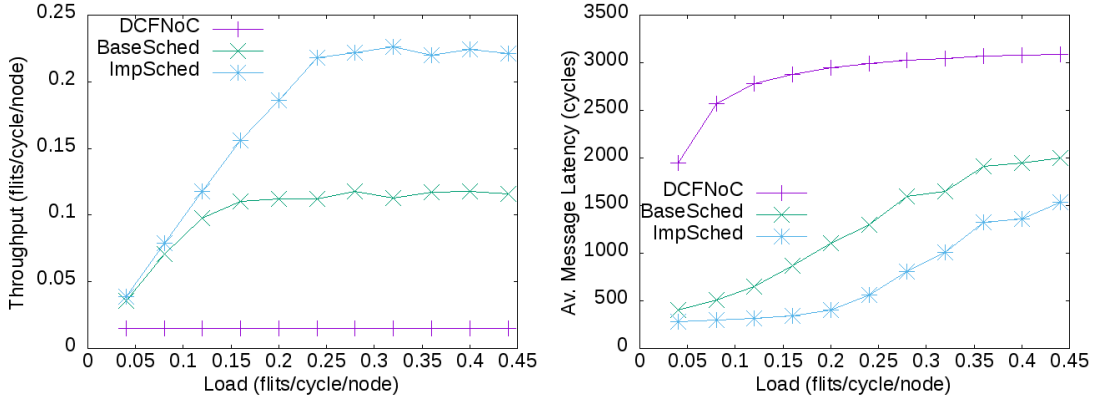
Impact of rescheduling. By implementing the rescheduling technique, the scheduler is able to maximize DCFNoC utilization reaching a significantly higher performance. Figure 5.13 and 5.14 show results for 4×4 and 8×8 mesh systems using DCFNoC without dynamic scheduler (*DCFNoC*), the baseline scheduler (*BaseSched*) and the improved scheduler that implements the rescheduling technique (*ImpSched*). For the 4×4 mesh scheduler we use 8 ways. Data window contains 16 slots (one per node). Figure 5.13a shows how network throughput has been improved from 0.062 (1/16) to 0.30 when using the dynamic scheduler reaching 0.43 flits/cycle/node with the *ImpSched*. In other words, the number of messages that can be transmitted per slot goes, on average, from 1 in DCFNoC to 5.8 in the *BaseSched* to 6.9 in *ImpSched*. For the *ImpSched* this results in nearly 110 messages per window. With respect to latency values Figure 5.13b shows how average message latency is kept low until 0.48 flits/cycle/node.

For a 8×8 mesh, the scheduler implements 16 ways at each node and data window



(A) (B)

FIGURE 5.13: Network throughput (a) and message latency (b) in a 4×4 mesh.

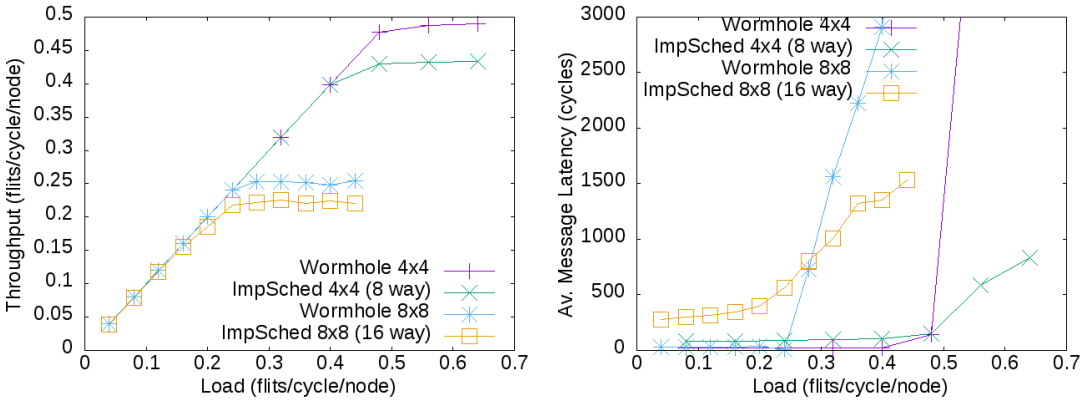


(A) (B)

FIGURE 5.14: Network throughput (a) and message latency (b) in a 8×8 mesh.

contains 64 slots (one per node). Before analyzing the results, it is important to remark that in a 8×8 mesh the throughput achieved per node is roughly divided by 2 with respect to a 4×4 mesh. Hence, network performance per node is expected to be reduced in a similar manner. Figure 5.14a shows network throughput for the 64-node mesh. DCFNoC obtains 0.015 (1/64), *BaseSched* gets 0.12 and the improved scheduler reaches 0.23 flits/cycle/node. These throughput numbers show how a network with 64 nodes is able to improve network capacity up to 14 messages per slot which doubles the capacity achieved in the 4×4 mesh. This means that the network is able to send nearly 900 messages every TDM window. The network keeps message contention low until 0.20 flits/cycle/node, hence Figure 5.14b shows how latency values are kept low.

hp-DCFNoC versus Wormhole. The goal of hp-DCFNoC is to improve the performance achieved with DCFNoC while keeping its valuable QoS properties. In this section



(A) (B)

FIGURE 5.15: Network throughput (a) and message latency (b) comparison versus standard wormhole in a 4×4 and 8×8 mesh.

we show how hp-DCFNoC is able to achieve that goal but also how its peak performance numbers are very close to the one of wormhole NoCs. Figure 5.15a compares the network throughput achieved by hp-DCFNoC with the one of a high-performance wormhole NoC. For the wormhole NoC we use a conventional NoC implementation with canonical and pipelined routers, one single virtual channel, round-robin arbitration, and XY routing. As shown in the plots, in both cases, the throughput of hp-DCFNoC is very close to the one that can be achieved with wormhole being close to 0.45 flits/cycle/node for a 16-node mesh and 0.22 for 64-node mesh. Interestingly, while wormhole NoCs have been able to provide performance guarantees [48] the latency bounds provided by these NoCs are much worse than the ones provided by hp-DCFNoC. Figure 5.15b shows how average message latency is kept low until 0.48 flits/cycle/node for a 16-node mesh and 0.22 for 64-node mesh.

5.2.4 Analyzing the Impact on Applications Behaviour

In order to evaluate the benefits of using hp-DCFNoC on applications execution time we have designed several synthetic kernels. The synthetic kernels generated produce a variable amount of instructions and the corresponding network messages targeting different destinations based on nature of the message. Instructions labeled as L1hit do not produce any network message. Instructions labeled as L1miss are injected into the network to a random destination and no further instruction is processed until the node receives a response. Messages originated from the L1miss instruction that are

TABLE 5.1: Different scenarios evaluated. We have modeled three main different scenarios for application traffic with four different levels of network congestion.

Scenario	L1hit	L1miss-L2hit	L1/L2miss-MC	Inj.Rate
1	95%	4%	1%	5%
2	95%	4%	1%	20%
3	95%	4%	1%	40%
4	95%	4%	1%	60%
5	80%	10%	10%	5%
6	80%	10%	10%	20%
7	80%	10%	10%	40%
8	80%	10%	10%	60%
9	70%	20%	10%	5%
10	70%	20%	10%	20%
11	70%	20%	10%	40%
12	70%	20%	10%	60%

labeled as L2hit do not produce additional traffic. However, messages labeled as L2miss produce an additional request to the memory controller (MC) and the node originating the request cannot progress until the response is received. To model requests processing time L2 requests and MC requests are also delayed at the destination by 10 and 40 cycles, respectively. L1hit requests are processed in one cycle. To evaluate the behaviour of this kernel under different levels of contention this traffic model is only executed at one node. The rest of the nodes inject random traffic at a specified load. All modeled scenarios are shown in Table 5.1.

We executed 5000 instructions in the considered scenarios when using hp-DCFNoC and wormhole NoCs. For these experiments we model a 4×4 NoC. The application traffic is executed in node 0 (top left-most node) and the MC is located at node 15 (bottom right-most node). Figure 5.16 shows the results of this experiment. As shown in the plot wormhole always provides higher throughput values. This is explained by the fact that average latency values of hp-DCFNoC are generally higher since a notification phase is required before transmitting the data. However, the performance differences are lower in the context of highly saturated scenarios. For these scenarios the network throughput provided by hp-DCFNoC is very close to the one provided by wormhole and zero load latency is less important. In particular, for scenario 1 hp-DCFNoC throughput is 20% lower than the one achieved by wormhole while in scenario 12 the difference is only 12%. It is also important to mention that these differences do actually represent a corner case since processor architectures usually include mechanisms to hide memory latency

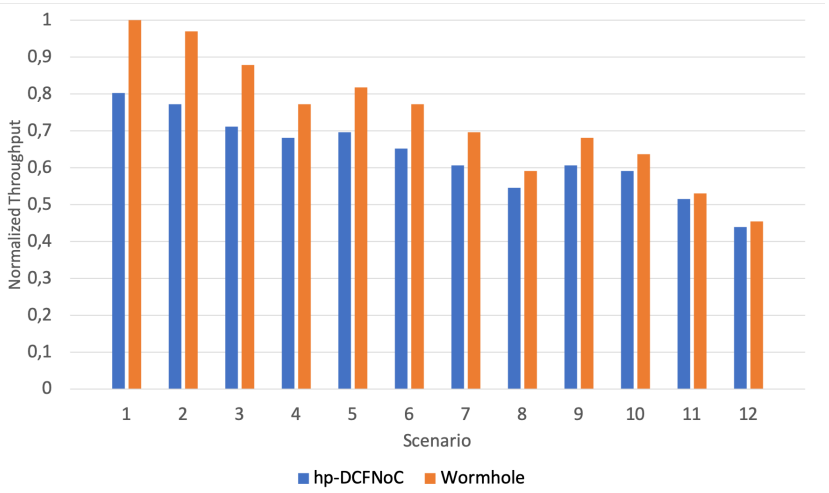


FIGURE 5.16: Normalized throughput comparison for different scenarios in a 4×4 mesh system when using hp-DCFNoC and wormhole NoCs. Application is originated traffic in the farthest node w.r.t to the MC.

like write-buffers, out-of-order execution, data and instruction prefetchers and the like. In our experiments for each instruction labeled as L1miss the injector is stalled until a response is received.

The obtained results also indicate hp-DCFNoC is well suited to applications with high-bandwidth requirements like the ones found in autonomous driving systems [70]. Note that performance guarantees of hp-DCFNoC are identical to the ones provided by DCFNoC while the performance that can be guaranteed in a 4×4 mesh wormhole NoC is much lower [48, 63].

5.2.5 Area Overhead and Frequency

Maximum operating frequency and area utilization are obtained using Cadence RC Compiler and the 45-nm Nangate library [60]. The scheduler implemented is *ImpSched* for a 4×4 and 8×8 mesh. The scheduler implements 8 and 16 ways at each node, respectively and data window contains 64 slots (one per node). The wormhole NoC implemented is 64-bit width and uses 8 slot input buffers with Stop&Go flow control. For the implementation results we consider two routers: the hp-DCFNoC notification ($Noti\_sw$) router and the data ($Data\_sw$) router for a 4×4 and 8×8 mesh. Data network configuration used includes 96-bit width links to implement a mesh network topology. Notification

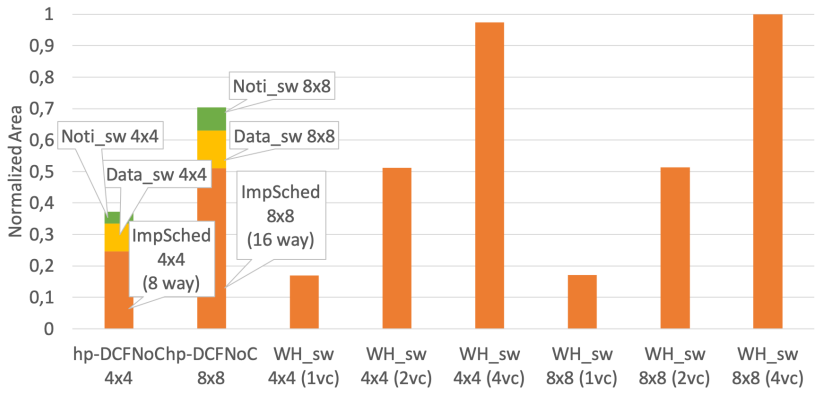


FIGURE 5.17: Area overhead at each node.

network implements 38-bit width links in this particular case. The wormhole router (*WH_sw*) used for comparison purposes implements (1, 2, and 4) virtual channels.

Figure 5.17 shows area overheads for every component when targeting high frequency. The 8×8 mesh hp-DCFNoC data router uses 30.42% less area than the WH-based one for a 8×8 mesh, when using one virtual channel (1vc), with a total area of $28,935\ mm^2$ and $41,273\ mm^2$, respectively. Wormhole router requires input buffers, flow control logic, routing units, output port arbiters and crossbar interconnect. On the other hand, the hp-DCFNoC router implements very simple routing logic in order to compute the output port, a crossbar interconnect as well as output delay registers. Due to small bit-width hp-DCFNoC notification router is the lightest router with a total area of $9,007\ mm^2$ and $17,715\ mm^2$ for a 4×4 and 8×8 mesh implementations, respectively.

However, the 8×8 mesh scheduler uses more area than a wormhole router for a 8×8 mesh with no virtual channel, although it uses roughly the same area compared to WH-based using 2 virtual channels with a total area of $123,551\ mm^2$ and $124,143\ mm^2$, respectively. It is important to remark that in general NoCs found in commercial processor require at least 2 VCs to avoid request and response messages deadlock and even more virtual channels when using cache coherence protocols to prevent protocol-induced deadlocks.

Figure 5.17 also shows total area of full hp-DCFNoC implementation including the *ImpSched*, notification and data routers. The 8×8 mesh hp-DCFNoC implementation uses 27% more area than the *WH_sw* for a 8×8 mesh with 2 VCs, however, is 42% lighter than a wormhole router when using 4 virtual channels.

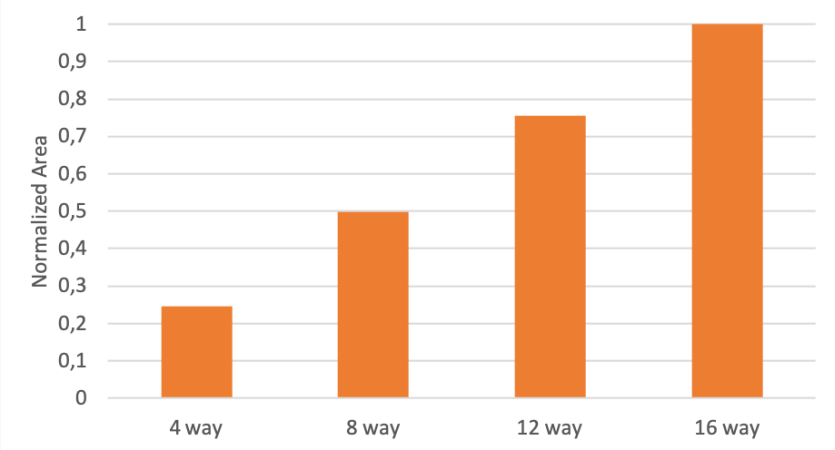


FIGURE 5.18: Area overhead for *ImpSched* implementations with different number of ways in a 4×4 mesh system.

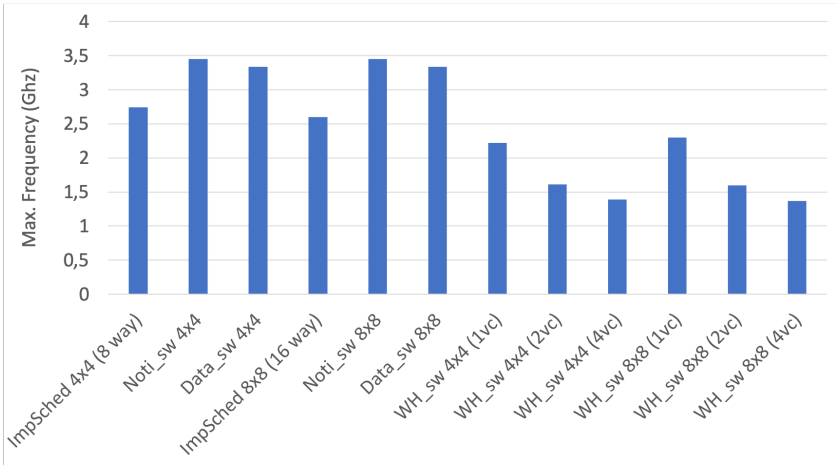


FIGURE 5.19: Maximum attainable clock frequency for all modules of hp-DCFNoC. Wormhole is also shown for comparison purposes.

Figure 5.18 illustrates how area overhead of hp-DCFNoC is affected as the number of ways increases. As we can seen the area overhead grows linearly as the *ImpSched* module implements more ways.

We have also analyzed the maximum attainable clock frequency of the different routers. Figure 5.19 shows that the simpler hp-DCFNoC routers design gets a significant boost in clock frequency by improving wormhole router's one by 55% and 50% for notification and data router, respectively. The critical path of the wormhole router limits clock frequency to 2.22 GHz. However, the hp-DCFNoC routers exhibits a critical path of 290 ps and 300 ps leading to a maximum clock frequency of 3.45 GHz and 3.33 GHz for notification and data router, respectively. Although wormhole router is more complex, it is pipelined in 4-stages which allows achieving high frequencies.

For hp-DCFNoC, in addition to the fast hp-DCFNoC routers we have a relatively complex scheduler that includes arbiters and route checkers. For the scheduling process we use only one cycle simplifying route notifications and control logic. Even with such critical path, hp-DCFNoC clock frequency reaches up to 2.74 GHz and 2.60 GHz for 4×4 and 8×8 mesh implementations, respectively.

5.3 Summary

Future safety-critical real-time systems will need processor designs able to provide performance guarantees without renouncing to peak throughput numbers. In this chapter we propose hp-DCFNoC, a new NoC design that satisfies this requirement by providing throughput numbers that are very close to the ones that can be achieved with standard best-effort wormhole NoCs. At the same performance level hp-DCFNoC is able to guarantee performance to applications with much less resources (area and power) than an standard wormhole implementation. To achieve the aforementioned demanding features hp-DCFNoC relies on a distributed scheduler built on top of DCFNoC that maximizes the number of packets that can be scheduled concurrently.

In this chapter we reached the main goal of this thesis. A NoC solution able to provide timing guarantees while enabling competitive performance. Nonetheless, in the next and final chapter we extend our work with a new view of alternative solutions derived from our gained experience.

Chapter 6

A Study on Conflict-Free TDM-based NoC Communications

In the context of real-time systems, the use of shared resources such as the NoC becomes challenging. Existing Hard real-time NoC platforms fundamentally provide static resource allocation, which does not fit the requirements of efficient platform management and does not take advantage on the context-sensitive nature of most real-time applications. More dynamism is in fact needed to flexibly allocate resources to real-time applications over time, while delivering strong isolation for security reasons. In this chapter, the focus is on analyzing different TDM conflict-free communication alternatives that ensure predictability and isolation between domains in a flexible manner. The main goal is to explore a wide range of network solutions trading off area for performance (especially, latency), while preserving domain isolation and time predictability.

This chapter explores different TDM-based real-time NoC solutions and proposes new TDM-based designs (Section 6.1). We graph and provide different area, performance and energy trade-offs. All configurations are analysed by inspecting the channel dependency graph. Finally, performance results are provided as well as area overhead and maximum attainable frequency in Section 6.2.

This chapter embeds the previous contributions of this thesis. Indeed, the development of DCFNoC enabled us to widen the set of possible solutions and to reach the definition of families of solutions based on the use of delays within the network.

6.1 Families of Solutions

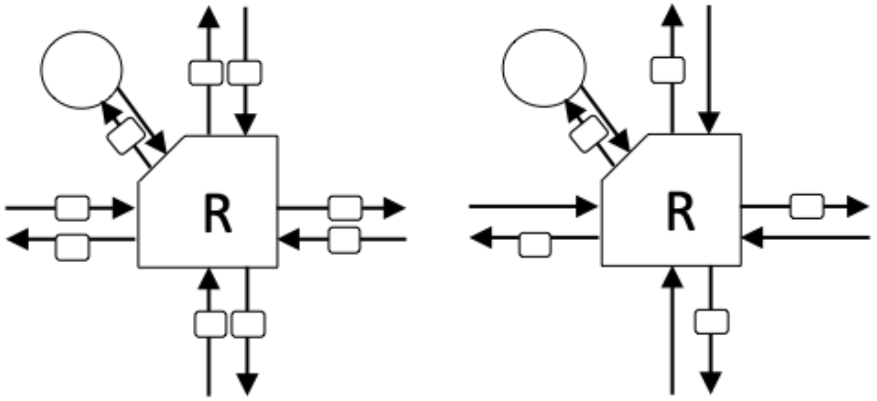
Since most of NoC internal resources are shared between packets from different domains, the NoC is key to preserve domain isolation. A domain is a set of communication flows belonging to an application or set of applications that must be isolated to avoid external contention or interference. Different domains can be used to isolate different critical applications with the same or different criticality level.

This section explores several TDM NoC solutions for conflict-free communications in safety-critical real-time systems. To do so, we analyse the channel dependency graph (CDG) of the network to identify potential conflicts and apply different methodologies to avoid them. Such methodologies revolve around the proper scheduling of communications in time and the selective insertion of propagation delays along network hops.

As before, our main goal is to achieve conflict-free transmission to allow synchronized scheduling commands through the network and to obtain a TDM-based implementation. The scheduling commands are just a means to orchestrate communications in time. In order to set the communication dependencies, we follow OSR [71] token propagation methodology throughout the CDG. OSR is a static reconfiguration mechanism for NoCs. This mechanism triggers a token from all end-nodes. Tokens advance through the network following the CDG to drain the network of packets and provide new configuration commands (routing function). It is important to remark that during this process, network traffic is not stopped. In this chapter we use the token propagation concept throughout the CDG as a way to find the TDM schedule that achieves conflict-free communications. We are not actually using token propagation but this concept is useful to enforce the message ordering along the CDG.

For the analysis, we group the identified solutions in three families, *Token propagation-based*, *DCFNoC-based* and *No-Delay*. The first family (Token), adds delays on selected network routers to synchronize packet transmission in the NoC. The second (DCFNoC), adds delays on I/O paths following the CDG in such a way that all network paths have the same length. The third (No-Delay), avoids adding delays by properly scheduling communications in time using a TDM Scheduler within the router injector.

We consider the following assumptions for all the families:



(A) (B)

FIGURE 6.1: Router architectures: (a) Two flip-flop router architecture (IOB), (b) One flip-flop router architecture (OB).

A1 *Communications are scheduled in time (TDM) by source nodes to all possible destinations.*

A2 *Ejection network bandwidth in every node is one flit per cycle. We assume a maximum network throughput of one flit per cycle.*

A3 *All TDM slots composing a period have the same length.*

A4 *We consider only single-flit messages. For larger messages TDM slots must be equal to message length.*

Additionally, two baseline router architectures are assumed in this chapter.

- **Input-Output buffered (IOB) router architecture** which implements a flip-flop at every input/output port (except input local). As Figure 6.1a shows, every router and link within the network have the same delay (one cycle each), in other words, two cycles per hop.
- **Output buffered (OB) router architecture** which implements a flip-flop at every output port. As Figure 6.1b shows, every router plus link pair within the network have the same delay (one cycle each), in other words, one cycle per hop.

TABLE 6.1: Summary of Concepts

Concept	**Description**
Token injector	Initial node where token starts since it has no input dependencies.
Supported domains (D)	Number of temporal conflict-free domains that can be on the fly in the topology.
Converging point	The point where the token propagation is sent from one output port and get back to the same port.
$Round - Trip\ Time\ (RTT)$	The number of cycles that takes the token to propagate from one router and to get back to the same router from another input port.
Relative latency	The difference in cycles from the token arrival time from one input port to backpropagate to the same router from another input port. This is always a multiple of RTT.
Global synchronization	Every router gets the same ID at each input port on every cycle.

6.1.1 Token Propagation-Based Family

This family of solutions relies on the token propagation approach. These solutions follow the principle of expanding the CDG to get the corresponding layered CDG, in which dependencies are ordered, but following the OSR token propagation methodology [3], as explained in more detail below. This is actually a key difference with respect to DCFNoC. For the shake of understanding the explanation, refer when needed to the summary of concepts given in Table 6.1.

With the OSR method, we aim to bring scheduling commands to all input ports of a generic NoC router in a 2D mesh. This is equivalent to ensuring and finding a TDM conflict-free scheduling in a particular NoC implementation. To do so, we associate TDM time slots with partitions/domains and add delays on selected network routers in such a way that every router serves the same domain from its input ports at the same time (see Figure 6.2). Tokens carry a domain identifier (DI), which identifies the corresponding domain. The domain determines which packets can be forwarded from a specific router input port when the token arrives. To achieve strong isolation between domains, token propagation needs to be synchronized along the network in a way that every router

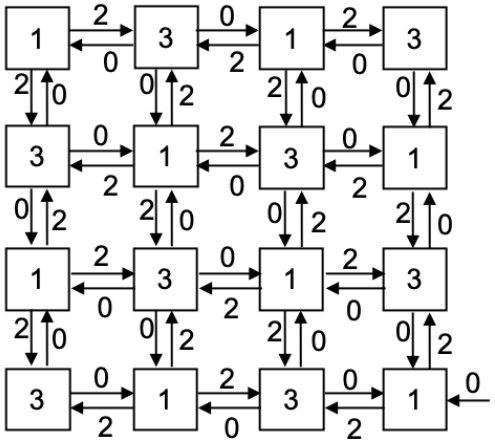


FIGURE 6.2: Token propagation flow with flooding methodology in a specific time slot for IOB router architecture. Numbers denote the token ID served on a specific NoC resource at the same clock cycle. The token is injected from the bottom right-most node.

gets the same ID at each input port on every cycle, see Figure 6.2. This means that a router will serve messages belonging to the same partition and that messages from different domains will never compete between them. This generates waves of same-ID slots throughout the network with minimum latency and conflict-free propagation (between different IDs). The resulting NoC is a bufferless and low-latency TDM implementation. This approach can be used with mainstream routing algorithms (e.g., XY, Segment-based Routing), or with more elaborated custom-tailored topologies and routing strategies to maximally exploit the benefits of this philosophy.

For the sake of understanding, this methodology can be broken down in the following steps:

Step 1 Compute the round-trip time (RTT) for the token propagation.

Step 2 Once the number of supported domains is known (RTT), configure the network to support such a number of domains.

Step 3 Define the TDM slot wheel assignment to manage message injection and avoid conflicts within a domain.

Step 4 Optionally, in order to support higher number of domains, add delays to enlarge the token propagation path.

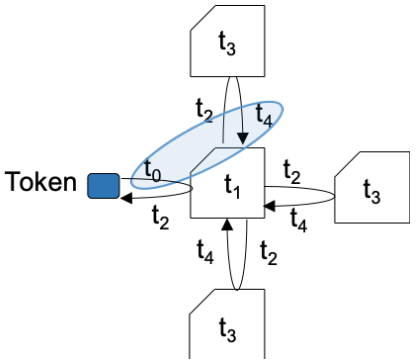


FIGURE 6.3: Token propagation round-trip time becomes a period of 4 cycles in IOB router architecture.

The following definitions generalize the token propagation methodology to obtain different custom-tailored topologies and routing strategies. We combine different solutions to fully exploit the benefits of this philosophy.

Definition 1. *The optimal number of supported domains in a 2D mesh can be determined by the shortest relative latency, that is equal to the number of cycles that takes the token to propagate from one router and to get back to the same router from another input port. We define this time as the round-trip time (RTT).*

Step 1. Let us consider an end node which injects a token to the network and this token propagates trough the network following a flooding methodology. Let us assume IOB routers. As Figure 6.3 shows, the router receives the token in time $t = 0$ and propagates the token through all output ports. Neighbour routers receive the token in $t = 2$ because every hop takes two cycles. Now, neighbour routers return token propagation as they also use flooding. Consequently, the first router receives the token back from all input ports in $t = 4$ resulting in $RTT = 4$. Indeed, Figure 6.2 shows a general view of a 4-domain token propagation through the network. Recall that token propagation needs to be synchronized along all the network and every router must receive the same ID at each input port on every cycle. With this, the router serves messages belonging to the same domain and enforces strong isolation between domains.

Definition 2. *In a 2D mesh, no matter the token propagation path is used, the Maximum Common Divisor (MCD) of all relative latencies in the network is equal to the shortest RTT.*

Step 2. Let us compute the MCD of a 2D mesh:

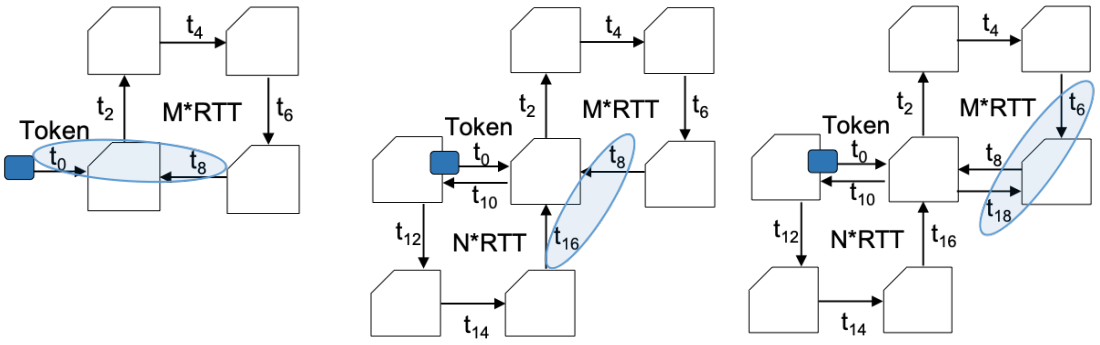


FIGURE 6.4: Token propagation using square shape in a 2D mesh using IOB router architecture.

In order to get a relative latency $> RTT$ we avoid the propagation shape of sending the token from one output port and getting back to the same port (converging point). Thus, we search for larger reconvergent paths. Let us bring it, without lack of generality, to an input test to the next available propagation shape that is a square (four routers).

As Figure 6.4a illustrates, one router receives a token from west and propagates the token to north. In $M * RTT$ cycles it receives the token form east, being M equal to 2. Now, the relative latency is $ABS\ (0 - 8) = 8$.

Next, with an intent to propagate in a square shape, we forward it to the south, see Figure 6.4b. Therefore, in order to reach this router again, we propagate the token to west and therefore, the token arrives back in $N * RTT$ cycles from the south, being N equal to 4. The absolute relative latency is:

$$Relative\ latency = ABS\ (M * RTT - N * RTT) = ABS\ (8 - 16) = 8 \tag{6.1}$$

Now, what is the next move? From this router we have already propagated the token to the north and also to the west, so the only available move is to propagate either to east or south. Let us propagate to east as depicted in Figure 6.4c. Once we have other converging points we need to compute the corresponding relative latency:

$$\begin{aligned} Relative\ latency = ABS\ ((M * RTT + N * RTT + 2) - (M * RTT - 2)) \\ = ABS\ (18 - 6) = 12 \end{aligned} \tag{6.2}$$

Now, we have to compute the MCD of all relative latencies following the token through the square path:

$$MCD\ (8, 12, 16, 20, 24) = 4 \tag{6.3}$$

As we can see, the MCD is 4 cycles being equal to RTT. This is because in a 2D mesh, no matter which token propagation path we use, the MCD of all relative latencies is always equal to the shortest round-trip time being 4 cycles in this case. Therefore, a 2D mesh has a natural period of 4 cycles, and hence, it supports 4 domains. Recall that every router must receive the same domain ID at each input port on every cycle. This means that we can associate 4 TDM time slots to 4 domains using the token propagation approach. We just need to apply the modulo operation of the corresponding token propagation cycle t_x $(mod\ 4)$ (see Figure 6.2). Finally, the number of cycles for any token propagation within the network is always multiple of RTT, and therefore global synchronization is guaranteed.

6.1.1.1 Conflict-Free Intra Domain

Step 3. In order to define the TDM slot assignment that guarantees conflict-free transmission within a domain, we also use the token propagation methodology. In a 2D mesh topology with IOB routers, every two nodes the token ID coincides, hence the network supports up to two message injections at the same time. Therefore, injected messages must be separated more than two cycles to avoid conflicts within a domain. To do that, we define a 16-slot wheel and assign different starting points in the TDM wheel to different node IDs to enforce two cycles of distance. For instance, a potential mapping that satisfies that condition (having two cycles of distance) is to classify nodes by ID (odd, and even). This is so because following XY coordinates, even or odd routers always are at a distance of two cycles. As Figure 6.2 shows, nodes with token ID 1 will start from slot 1 and nodes with token ID 3 will start from slot 3, see Figure 6.5. Every cycle the node slot pointer moves one position forward. In order to better spread the messages without using additional logic we use the node ID to trigger the message injection when this is equal to the TDM slot ID. At the end of the wheel every node has injected one packet, so we achieve an average injection of one flit per cycle.

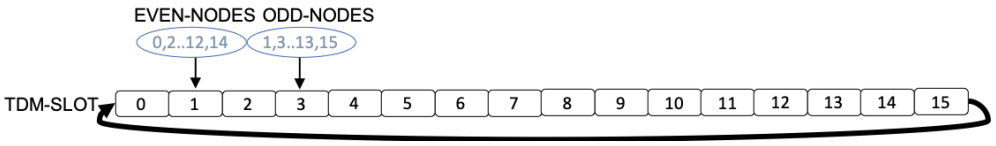


FIGURE 6.5: TDM slot wheel assignment to manage message injection in a 2D mesh with support for 4 domains. Every node is allowed to inject when the slot ID is equal to its node ID. Allowed inject nodes are represented by red circles at TDM slot wheel. Messages are represented by inject node numbers and subindex represents the propagation time in cycles.

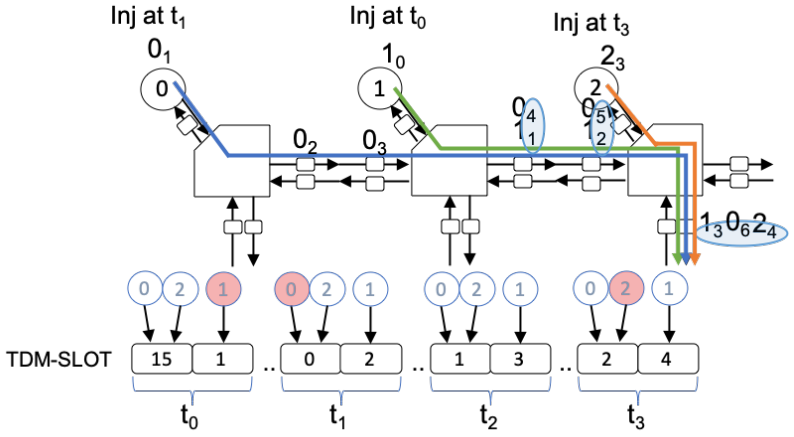


FIGURE 6.6: Scenario 1, consecutive nodes in the same row injecting to the same X direction. Message injection mechanism in Period 4 solution based on TDM slot wheel assignment using IOB routers. Nodes highlighted in red are the ones injecting.

For the sake of understanding, we show four different scenarios to illustrate the message injection mechanism taking into account the TDM slot assignment. Figure 6.6 illustrates scenario 1 where consecutive nodes in the same topology row inject messages in the sense that there is one or more shared links along those message paths. In this particular case, the assigned slot ID to node 1 at TDM slot wheel is 1, hence node 1 is able to inject at t_0, represented by a red circle in the TDM slot wheel. The message from node 1 will reach the output port of router ID 2 at t_3 (represented by 1_3). Later, node 0 will inject the message at t_1 and will reach the output port of router ID 2 at t_6 (represented by 0_6). Finally, node 2 injects the message at t_3 and will reach the output port of router ID 2 at t_4 (represented by 2_4). By observing Figure 6.6 we can confirm the messages arrival time (t_x) at every shared link along the path. As we can see, messages arrive in different time, thus this case is conflict-free.

Figure 6.7 shows scenario 2 where nodes at the topology corner inject messages using shared links. In this specific case, the assigned slot ID to node 13 at TDM slot wheel

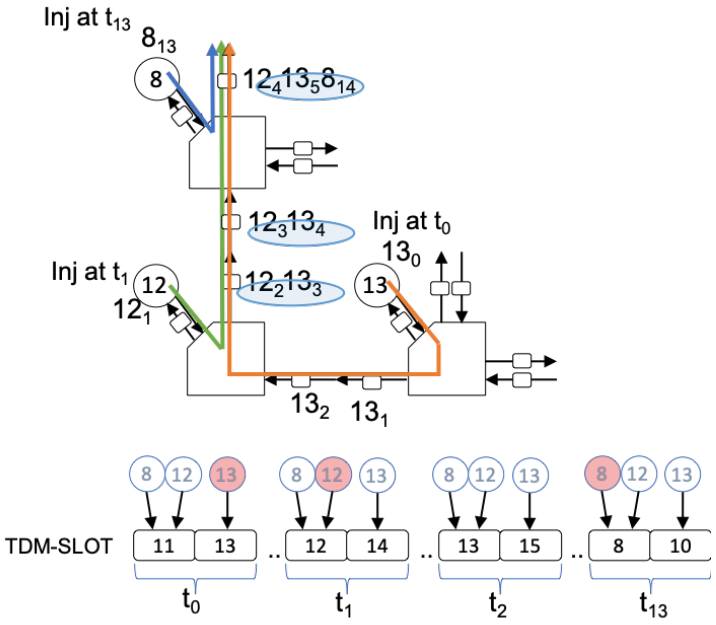


FIGURE 6.7: Scenario 2, corner nodes injecting to the north.

is 13, hence node 13 is able to inject at t_0, represented by a red circle in the TDM slot wheel. The message from node 13 will reach the output port of router ID 8 at t_5 (represented by 13_5). Later, node 12 injects a message at t_1 and will reach the output port of router ID 8 at t_4 (represented by 12_4). Finally, node 8 will inject the message at t_{13} and will reach the output port of router ID 8 at t_{14} (represented by 8_{14}). As we can see, messages arrival time at every shared link is different, hence this case is conflict-free.

Figure 6.8 illustrates scenario 3 where consecutive nodes in the same topology row inject messages to different X direction in the sense that message paths use shared links. In this particular case, the assigned slot ID to node 13 at TDM slot wheel is 13, hence node 13 is able to inject at t_0. The message from node 13 will reach the output port of router ID 14 at t_3 (represented by 13_3). Later, node 15 injects the message at t_2 and will reach the output port of router ID 12 at t_5 (represented by 15_5). Finally, node 14 will inject the message at t_3 and will reach the output port of router ID 14 at t_4 (represented by 14_4). As we can see, messages arrival time at the shared link of router 14 is different, hence this case is conflict-free.

Figure 6.9 shows scenario 4 where consecutive nodes in the same topology column inject messages to the south using shared links along those message paths. In this concrete case, the assigned slot ID to node 0 at TDM slot wheel is 0, hence node 0 is able to

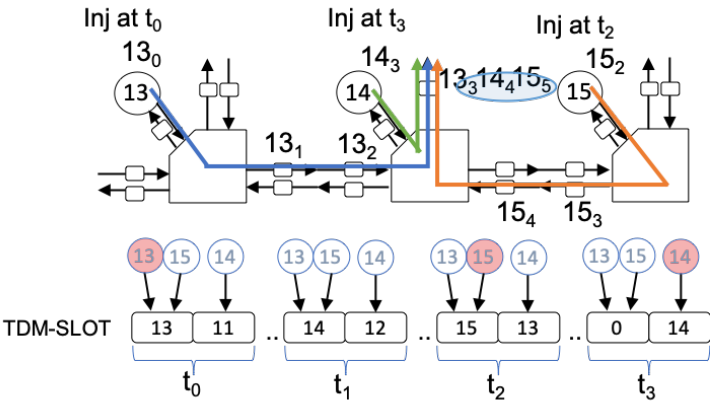


FIGURE 6.8: Scenario 3, consecutive nodes in the same row injecting to different X direction.

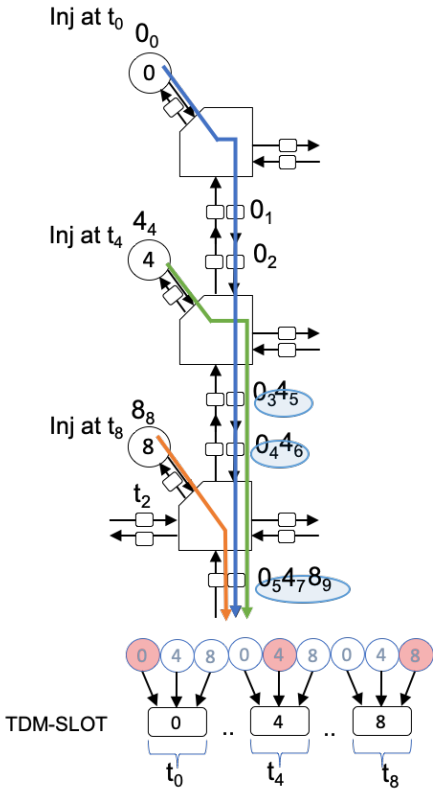


FIGURE 6.9: Scenario 4, consecutive nodes in the same column injecting to the south.

inject at t_0. The message from node 0 will reach the output port of router ID 8 at t_5 (represented by 0_5). Later, node 4 injects the message at t_4 and will reach the output port of router ID 8 at t_7 (represented by 4_7). Finally, node 8 will inject the message at t_8 and will reach the output port of router ID 8 at t_9 (represented by 8_9). As we can see, messages arrival time at every shared link is different, hence this case is conflict-free.

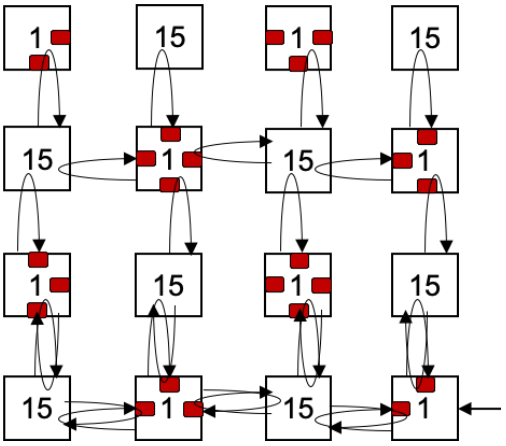
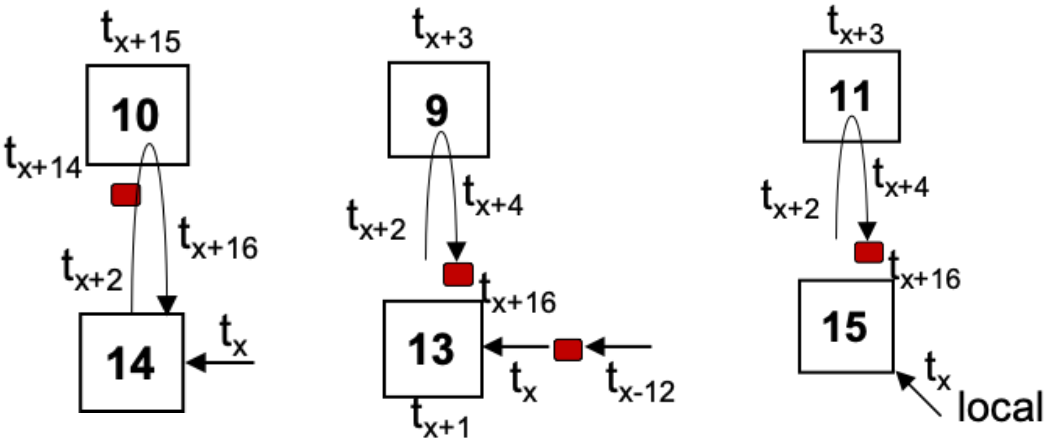


FIGURE 6.10: Flooding propagation in a 2D mesh using IOB routers composing a period of 16 cycles. This is a Period 4 solution which have a natural period of 4 cycles and uses additional delays (12 cycles in every red square) to enlarge the period to 16 cycles. Additional delays are represented by red squares at input ports. Some links are not plotted for representation purposes.

6.1.1.2 Increasing the Number of Domains

Definition 3. *In a 2D mesh topology with IOB routers, with $D-4$ additional delays every two hops along the flooding propagation path, the conflict-free synchronized property is preserved.*

Step 4. We have the following situation that is affected by all round-trip times, see Figure 6.10. In this figure we show the Period 4 solution which has a natural period of 4 cycles and uses additional delays to enlarge the period to 16 cycles, and hence to support 16 domains. The token is propagated using the flooding methodology along the network. Some links are missing for representation purposes. In this figure, we have three different situations all affected by the round-trip token described in Figure 6.11a. Note that additional delays are placed at input ports. Figure 6.11a shows a situation in which a router without additional delays receives the token in $t = x$. This router propagates the token and arrives to the next input port at $t = x+2$. This input port has twelve additional delays and the token arrives at router in $t = x + 14$ and is propagated back arriving at $t = x + 16$. Another different situation, depicted in Figure 6.11b, is when a router with additional delays receives a token to an input port with additional delays in $t = x - 12$. Token arrives at router in $t = x$ and is propagated to the next router arriving there at $t = x + 2$. Later, this router propagates back the token and arrives back at $t = x + 16$.



(A) (B) (C)

FIGURE 6.11: Breaking round-trip time limits by additional delays to support more domains. Additional delays are represented by red squares.

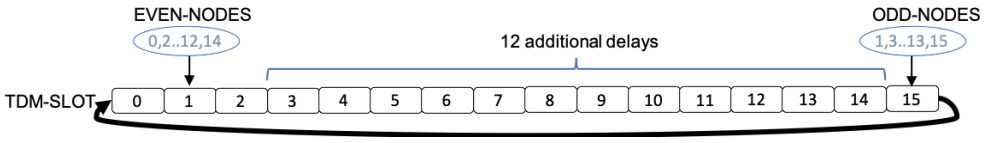


FIGURE 6.12: TDM slot wheel assignment to manage message injection in a 2D mesh with support for 16 domains. Every node are allowed to inject when the pointed slot ID is equal to its node ID.

Figure 6.11c shows the last situation when a token injector router without additional delays starts propagation in $t = x$ and arrives to the next router at $t = x + 2$. Later, the next router propagates back the token and arrives at $t = x + 4$ to an input port with additional delays. It takes twelve additional cycles to arrive to the router and the arrival times becomes $t = x + 16$.

As we can see, we have incremented the round-trip time to $RTT = 16$ cycles in all possible situations, and now we can support 16 domains.

To define the TDM slot assignment we need to take into account the 12 additional delays. Starting from Figure 6.5, as the round-trip time has increased 12 cycles, we move the odd nodes pointer 12 positions forward, see Figure 6.12.

Detailed implementation of this token propagation family as well as additional solutions are described in Appendix A.

In general terms, these solutions consist of customizing the 2D-mesh topology to support larger number of domains by construction as well as for large network size. To do so, the following techniques are applied:

- In order to support higher number of domains without adding delays, we remove links to build a topology with combined and overlapped rings. **Solutions derived:** Period 8 and Period 16 for a 4×4 mesh.
- In order to support larger networks, we combine larger unidirectional and overlapped rings with and without adding delays. **Solutions derived:** Period 32, Period 64 and Period 16 for a 8×8 network.
- In order to analyse the impact of varying the location of those additional delays, we spread delays along the network adding router stages. **Solutions derived:** Token 1 STAGE, Token 2 STAGES and Token 3 STAGES for a 4×4 topology.

These solutions, although described in the Appendix A, they will be compared to other solutions in this chapter.

6.1.2 DCFNoC Based Family

DCFNoC is proposed in Chapter 3. DCFNoC follows the routing algorithm to layer the CDG. Starting from a layered CDG the philosophy is to add delays on I/O paths in such a way that all paths have the same length to be able to serialize communications. For comparison purposes, we describe the DCFNoC methodology following the OSR token propagation concept. Recall that there is no token propagation, but we use this concept to order and expand the CDG and derive the CDG_l. In fact, we realize that the OSR token propagation methodology with XY routing algorithm obtains equivalent results to the initial DCFNoC approach proposed in Chapter 3. For DCFNoC family, we analyse two different solutions, one following DOR (XY) routing algorithm and another one following Segment-based Routing algorithm (SR) [32].

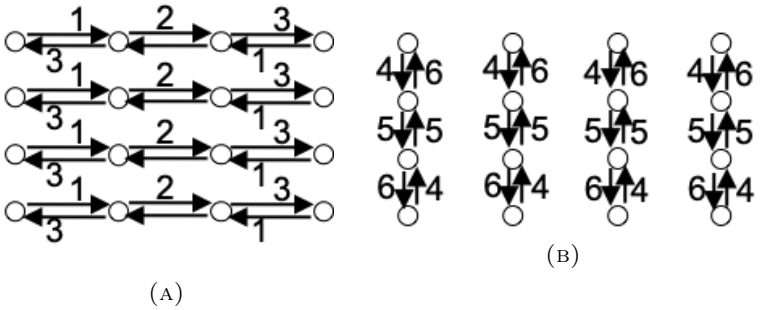


FIGURE 6.13: Initial DCFNoC approach link order following XY routing algorithm in a 4×4 mesh. Nodes are represented by circles and propagation layers are represented by arrows. Numbers represents every propagation layer thorough the CDG_{dl}.

6.1.2.1 DCFNoC Following XY Routing Algorithm

In DCFNoC with XY routing algorithm, a packet has to take into account all input dependencies before leaving a router. So, we cannot use a flooding methodology to propagate scheduling commands, since flooding does not fit very well with dependencies of the packets and would break the top-down layered CDG rule. Remember that, channels are assigned to only one particular layer $L\#$ in the CDG_l.

For comparison purposes, Figure 6.13 shows the initial DCFNoC approach link order following XY routing algorithm in a 4×4 mesh. As we can see in a 4×4 network we use six layers to traverse the whole network. Recall that the number of layers is defined by the diameter of the network (i.e. the longest minimal path) plus the injection and ejection links.

Figure 6.14 shows OSR token propagation methodology following XY routing algorithm to build a layered CDG. Token starts from corner routers since they have no input dependencies. Initially, X dimension is traversed from phase 1 to 3. Later, Y dimension transitions start gradually from phase 3 to phase 6 when all X dimension dependencies are solved. As we can see, initial DCFNoC approach and the OSR token propagation methodology both use six layers to traverse the whole network, hence results are equivalent.

Based on Figure 6.14 token propagation phases, we use the propagation phase labels to assign channels to a particular layer $L\#$, and construct the delayed layered CDG illustrated in Figure 6.15. This is a subgraph representing a CDG_{dl} from all nodes to node 15. Delays are positioned strategically following XY token propagation phases.

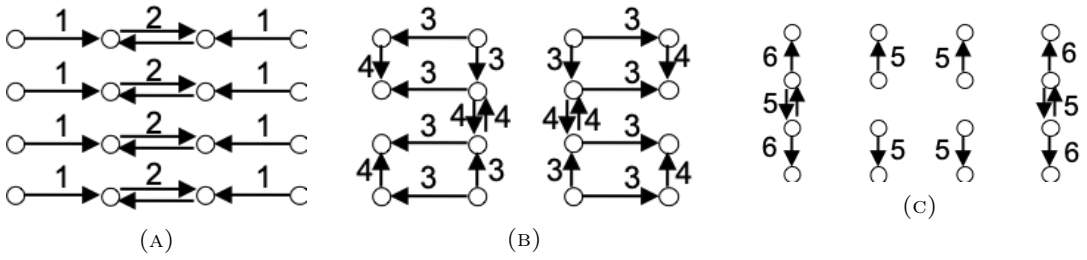


FIGURE 6.14: OSR token propagation phases following XY routing algorithm in a 4×4 mesh using OB routers. Nodes are represented by circles and OSR token propagation phases are represented by arrows. Numbers represents every propagation phase.

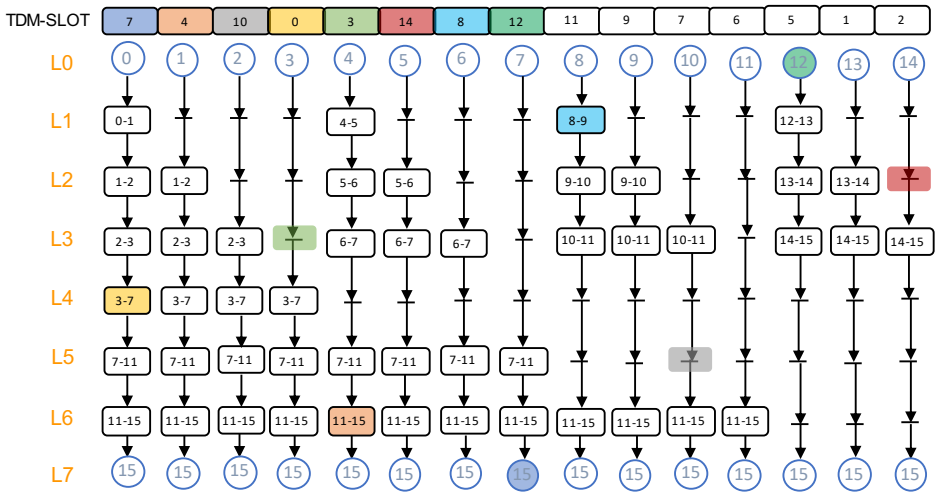


FIGURE 6.15: CDG_{dl} for the 4×4 mesh topology using OB routers and the XY routing algorithm.

In Figure 6.15, every TDM slot is associated to an injector ID. There is also a color representation to illustrate a packet position following this TDM slot assignment. Note that every row represents a network layer in the CDG_{dl}. Since only one injector per TDM slot is allowed to inject packets, we enforce one packet per layer rule to avoid conflicts. As we can see, TDM slots can be arbitrarily associated to injectors since messages from other TDM slot will never compete for the same channel at the same time.

6.1.2.2 DCFNoC Following SR Routing Algorithm

Another way to achieve isolation in NoCs is to use Segment-based Routing algorithm (SR) [32]. SR splits the network in disjoint sets of interconnected routers and links called partitions. Some routing restrictions are placed to break dependency cycles. These routing restrictions avoid packets to take some turns to preserve deadlock freedom (see

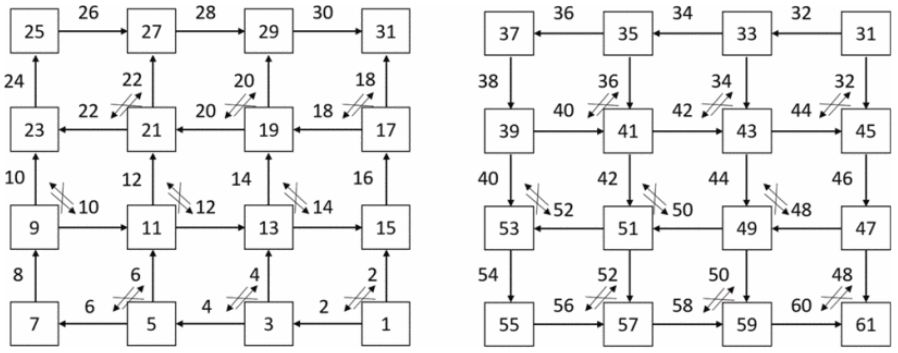


FIGURE 6.16: Network-level token propagation, with annotated latency, in the order of the CDG with periodic SR routing (dictating the position of routing restrictions) and single-cycle routers and links. Source [3].

Figure 6.16). In order to construct the layered CDG, we follow OSR token propagation methodology with SR routing to construct the layered CDG.

Figure 6.16 shows token propagation methodology following SR routing through the CDG. Tokens are triggered from the bottom-right corner as proposed in [3]. Bidirectional routing restrictions are placed to break potentially dependency cycles avoiding packets to take some turns to preserve deadlock freedom. Based on Figure 6.16, we construct a layered CDG illustrated in Figure 6.17.

Injection nodes represented as blue circles are placed at the top. The SR routing algorithm, following routing restrictions, determines the path from injection nodes to bottom destination nodes. Since every packet needs to use its path, links following a top down layers order, every layer represents one hop in time. Red-dashed arrows shows potential conflicts when using non consecutive links in terms of layers, thus we need to introduce the same amount of delays as layers we need to skip. Since all injection nodes are in L0 and all destination nodes in last layer, all paths have the same length. This solution involves all possible paths, hence is the worst case solution.

6.1.3 No-Delay Based Family

This family of solutions avoids adding delays by properly scheduling communications in time (i.e using a TDM Scheduler). The philosophy is based on the principle of packets travelling over the network will not collide with new injected packets as those injections will be delayed. First, we apply this method to mainstream routing algorithms (e.g. XY). Later, we adapt the routing algorithm to maximize throughput.

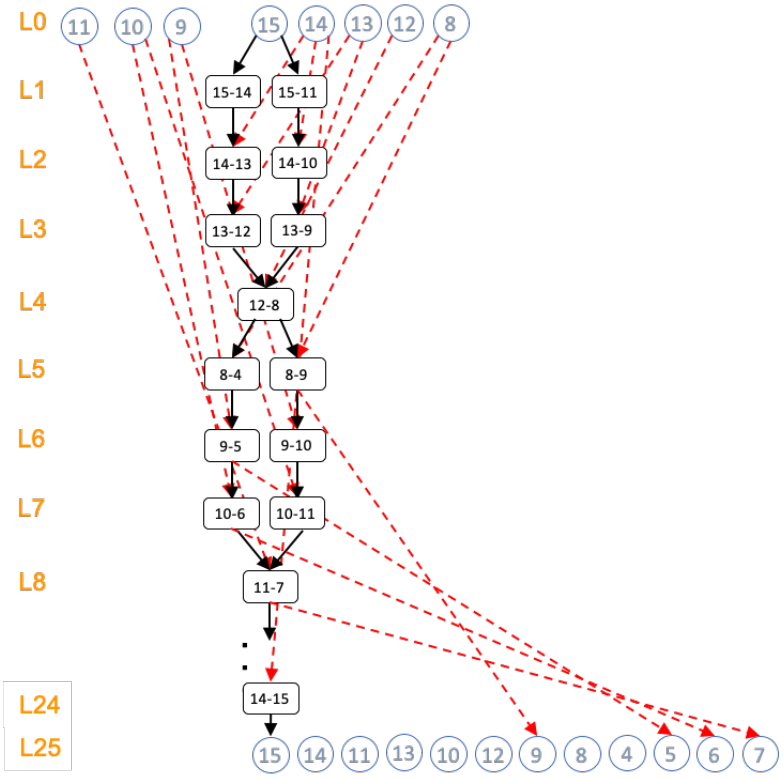


FIGURE 6.17: CDG_l for the 4×4 mesh topology using OB routers and the SR routing algorithm.

6.1.3.1 Couples Injection Following XY Routing

In this approach, the packet travelling time determines the injection time of next couple of injectors. To do so, we need to determine which is the packet travelling time between each couple of injectors along the network. In this solution, a minimal path routing algorithm is used (e.g. XY). For the sake of explanation we consider the following scenario, see Figure 6.18. This scenario would help us determining the packet travelling time between couples of nodes with IDs (0,1), (4,5) and (8,9). As we can see, couples of nodes (0,1) inject at t_0 (represented by 0_0 and 1_0). Those nodes are propagated along the network following a broadcast routing. This would help us to check packets travelling time at every point in the network. The message from node 0 will reach the output port of router ID 5 at t_5 (represented by 0_5). Later, a message from node 1 will reach the output port of router ID 4 at t_5 (represented by 1_5). Therefore, the packet travelling time between couples of injectors (0,1) and (4,5) is 5 cycles using the IOB router architecture. In addition, the couple of nodes (4,5) injects at t_5 (represented by 4_5 and 5_5). The message from node 4 will reach the output port of router ID 9 at t_{10} (represented by 4_{10}). Later,

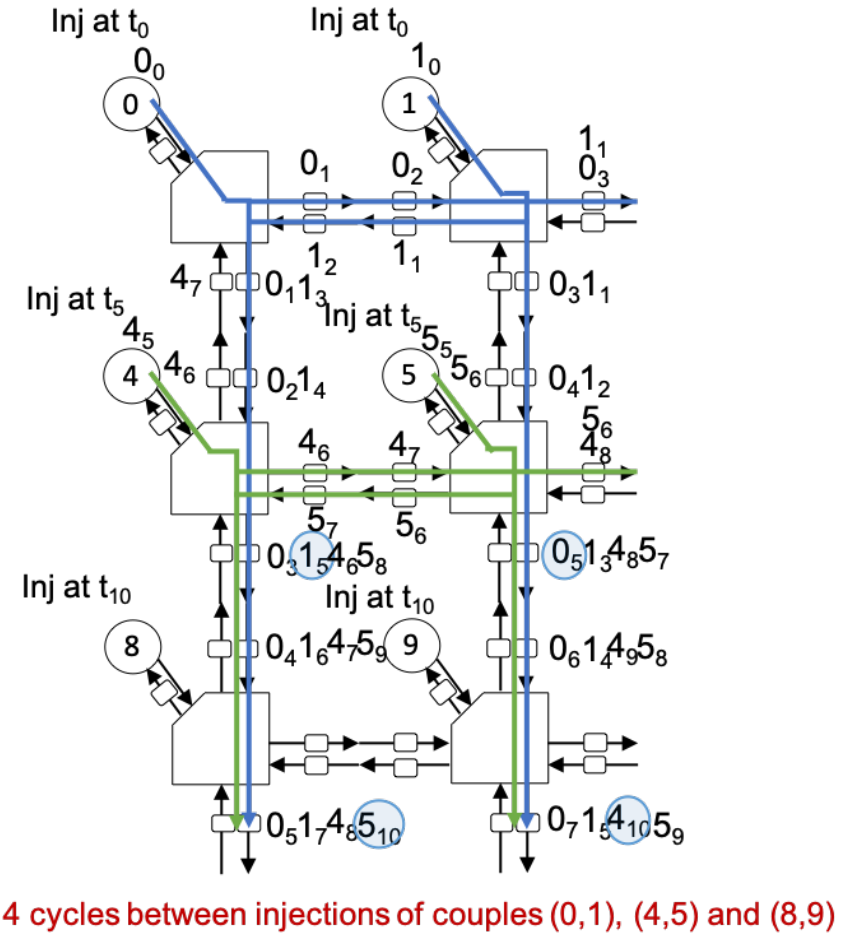


FIGURE 6.18: Determining the packet travelling time between couples of injectors to avoid conflicts with new injected packets. For clear representation purposes, some packet travelling time are missing.

a message from node 5 will reach the output port of router ID 8 at t_{10} (represented by 5_{10}). So, the injection time of pair of nodes (8,9) should be at t_{10}.

The next injection time of a pair of nodes must be computed following this methodology taking into account that current injected packets must overtake future couple of injectors.

Figure 6.20 shows worst-case relative latencies between consecutive couples of injectors. Assuming that one hop takes two cycles and starting by injectors $(0, 1)$, the worst-case to overtake the next injectors $(4, 5)$ are two hops (4 cycles). This means that following minimal routing and after 2 hops, packets belonging to injectors $(0, 1)$ will never compete between packets of injectors $(4, 5)$ since these router positions have been overtaken by all possible paths of $(0, 1)$ packets. Figure 6.21 shows a CDG of injection nodes $(0, 1)$. All possible links (represented by squares) are shown for injectors $(0, 1)$ but only some

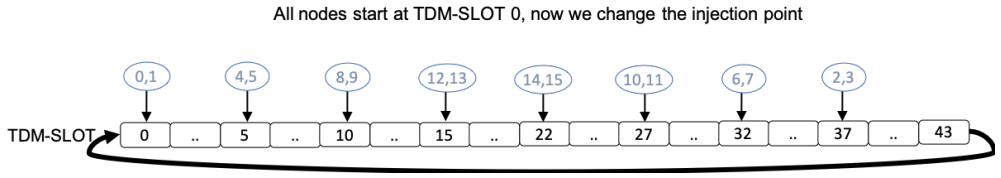


FIGURE 6.19: TDM slot wheel assignment to manage couples message injection in a 2D mesh using IOB routers. Every node are allowed to inject in the pointed slot ID.

ejection nodes are shown for better representation. Since all links are shown, we can realise that worst-case messages belonging to injectors $(0, 1)$ will be at router 5 (for injector 0) and 4 (for injector 1) at layer 2 (L2), that is the same, 2 hops afterwards. Therefore injectors $(4, 5)$ are able to inject 2 hops after $(0, 1)$ injection time.

In order to determine the TDM slot assignment we define a 44 slots wheel in which all nodes have the same starting point, but the trigger point is changed to preserve relative latencies between couples of injectors. As Figure 6.20 shows, nodes $0 - 1$ will inject at slot 0 and nodes $4 - 5$ will inject at slot 5, see Figure 6.19. Every cycle the node slot pointer moves one position forward. At the end of the wheel, every node has injected one packet, so we achieve an injection of 16 messages in 44 slots (lower than one flit per cycle).

Going back to Figure 6.20 we can see that relative latencies between couples of injectors in the same column is 4 cycles (2 hops) and between different column is 6 cycles (3 hops). As a result, in Figure 6.20 injection time of every injector is represented by numbers at routers. This approach schedules communications in time and avoids adding delays, but TDM period is large.

6.1.3.2 Diagonal Following Custom Routing

To avoid large TDM periods with this family we introduce a new solution. This approach follows the principle of overtaking future injectors but in this case we try to increase the number of current injectors to improve the TDM period. To do so, we customize the routing algorithm. Additionally, we serialize all messages in a unique router by enforcing all messages to cross this router (router 15 in a 4×4 mesh). A serialization router must be positioned in a topology corner to avoid having lots of dependencies.

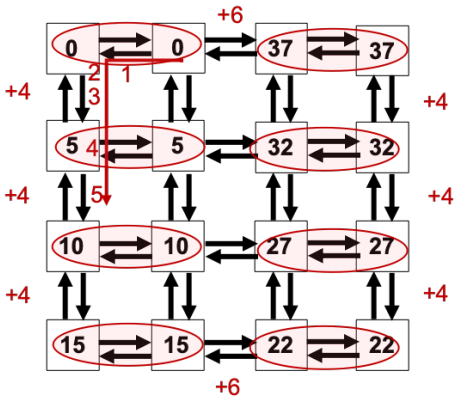


FIGURE 6.20: Couples injection solution. Numbers at routers represent nodes injection time. Relative difference between injectors by couples.

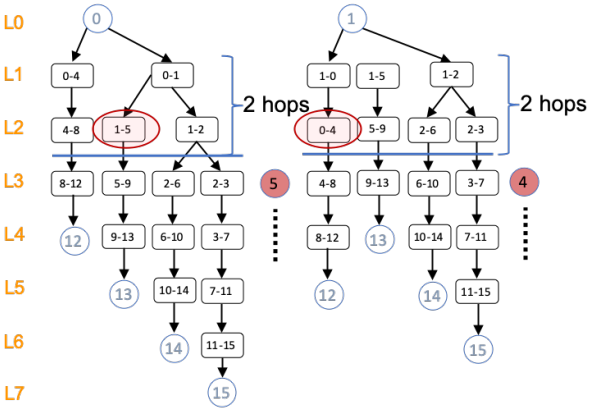
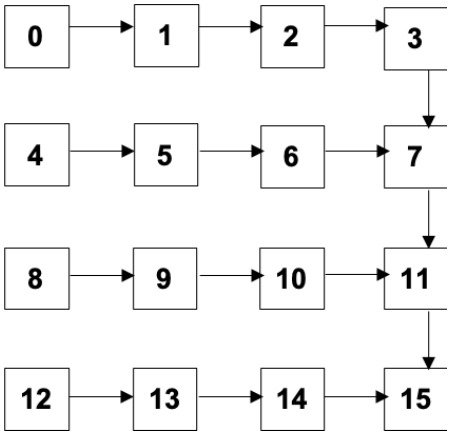
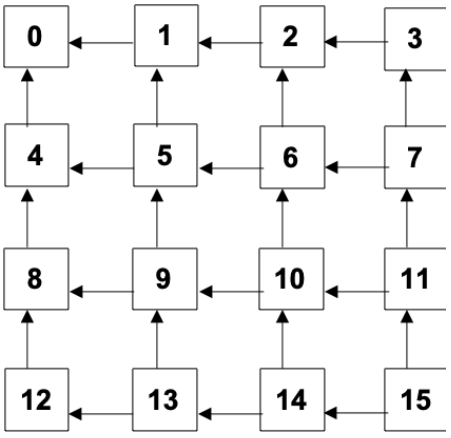


FIGURE 6.21: Couples injection CDG of router 0 and 1. Relative difference between injector nodes $(0, 1)$ and $(4, 5)$ are two hops.



(A) First routing phase, route to serialization router (router 15).



(B) Second routing phase, route to destination node.

FIGURE 6.22: Diagonal approach. Improving TDM period by using more injectors at the same time (columns) as well as custom routing to serialize packets in one point (router 15). Since first phase and second phase are using different network links conflicts are avoided. Destination nodes are only reached at second phase.

Initialization is triggered by columns, so all routers of the same column are able to inject at the same time. In order to serialize the messages belonging to each router, we use the bottom right-most router, thus first phase routing consist of using XY routing to reach this serialization router, as Figure 6.22a depicts. Once a message reaches a serialization router, conflicts are avoided and the message is able to use second phase routing to reach destination router, see Figure 6.22b.

For the sake of explanation we consider the following scenario, see Figure 6.23. This

scenario would help us to determine the packet travelling time between two columns by analysing packets from third column injector nodes. As we can see, the third column nodes (2,6,10,14) inject at t_0 (represented by 2_0, 6_0, 10_0 and 14_0). Those nodes are propagated along the network to bottom right-most router (15). The message from node 14 will reach the input port of router ID 15 at t_2 (represented by 14_2). Later, message from node 10 will reach the input port of router ID 15 at t_4 (represented by 10_4). Next, the message from node 6 will reach the input port of router ID 15 at t_6 (represented by 6_6). Finally, the message from node 2 will reach the input port of router ID 15 at t_8 (represented by 2_8). Therefore, the packet travelling time between two columns of injectors in a 4×4 mesh is 8 cycles using the IOB router architecture. So, the injection gap between two consecutive columns of injectors should be 8 cycles.

Regarding initialization and relative differences between columns of injectors at Figure 6.24, we compute the worst-case path between packets belonging to the third column (critical path) after reaching serialization node (node 15). Starting by the third column injectors $(2, 6, 10, 14)$, the worst-case to overtake the serialization router is four hops (8 cycles). This means that following minimal routing and after 4 hops, packets belonging to third column injectors will be serialized each other since they arrive at serialization router in different time. Therefore, at Figure 6.25, numbers at injection time of every injector are represented by numbers at routers.

In order to determine the TDM slot assignment, we define a 32 slot wheel in which all nodes have the same starting point, but change the trigger point to preserve relative latencies between couples of injectors. As Figure 6.25 shows, first column nodes $(0, 4, 8, 12)$ will inject at slot 0 and second column nodes $(1, 5, 9, 13)$ will inject at slot 9, see Figure 6.26. Every cycle the node slot pointer moves one position forward. At the end of the wheel every node has injected one packet, so we achieve an injection of 16 messages in 32 slots (lower than one flit per cycle).

This approach improves TDM period but the resulting period is still too long. Ideally, we need a TDM period equal to the maximum number of domains in the topology thus, equal to the number of routers. To do so, as explained in next section, we need to fill routers pipeline stages. By filling routers pipeline stages the diagonal approach is able to inject 32 messages in 32 slots, hence one flit per cycle.

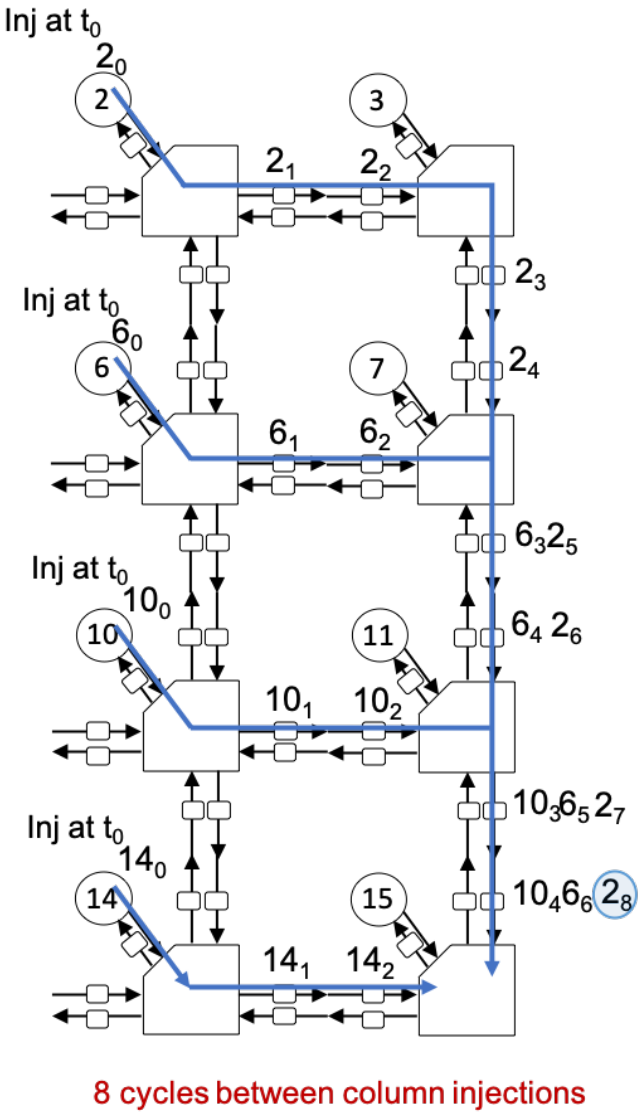


FIGURE 6.23: Determining the packet travelling time between two columns of injector nodes to avoid conflicts with new injected packets.

6.1.3.3 Filling Router Pipeline Stages to Improve TDM Period

To improve the network throughput we need to achieve a TDM period equal to the maximum number of domains in the topology. Since our IOB router contain two flip-flops there is one pipeline stage. One method to improve the TDM period when we have a router architecture with pipeline stages is to fill as many router stages as we have by injecting consecutive packets. By doing this, we avoid bubbles between packets, hence the number of injected packets is equal to the TDM period resulting to a network throughput of one flit per cycle.

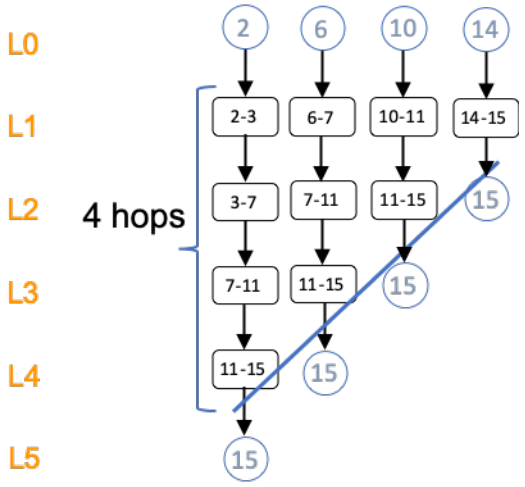


FIGURE 6.24: Columns injection *CDG* of third column (worst-case). Relative difference between third column injector nodes $(2, 6, 10, 14)$ and serialization node 15 are four hops.

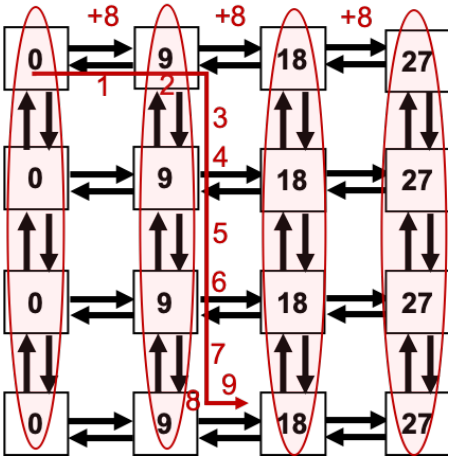


FIGURE 6.25: Relative difference between injector nodes by columns

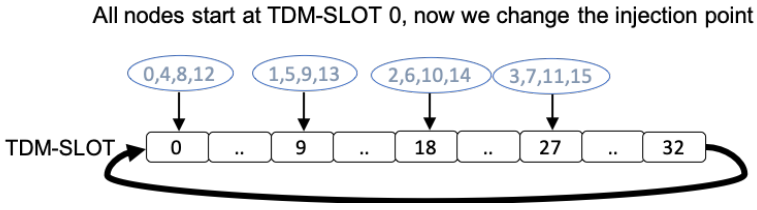
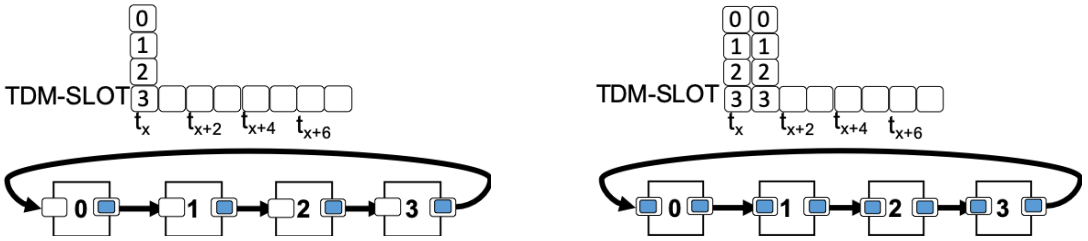


FIGURE 6.26: TDM slot wheel assignment to manage columns message injection in a 2D mesh using IOB routers. Every node are allowed to inject in the pointed slot ID.



(A) (B)

FIGURE 6.27: Fulfilling pipeline stages of a 4 router ring with a period of 8.

Figure 6.27a shows a ring of 4 routers with a period of 8. All of them are able to inject in $t = x$ and then have to wait until whole period ends. This means that we are injecting 4 messages every 8 cycles. To fill one pipeline stage of our router we can inject messages consecutively in two time slots, see Figure 6.27b. Then, we fill the bubbles between packets, hence injecting 8 messages every 8 cycles. This technique is applied in TOKEN RING and DIAGONAL solutions using IOB routers.

6.2 Evaluation Results

In this section, we compare the solutions of using IOB and OB router architectures. Scalability results are provided to evaluate which solution scales better and in which aspect. Additionally, in order to analyze the feasibility of implementing the proposed solutions, we provide area and maximum attainable clock frequency for the most competitive routers. Finally, we present load latency results to determine which solution is more efficient.

6.2.1 Experimental Setup

We design all investigated solutions using verilog RTL which can be synthetized for FPGAs and ASIC. We simulate the system using the Xilinx Vivado [58] RTL simulator. Thus, results presented in the chapter match exactly the number of cycles of a potential implementation.

The initial target topology is a 4×4 2D-mesh, and its derivatives obtained by removing selected links. The goal is to support 16 domains with all solutions. The network is fed by a message system generator implemented at each network interface using uniform traffic pattern. In order to create uniform traffic pattern, we use a pseudo random number generator with Linear Feedback Shift Registers (LFSR [69]) to generate a random destination label for every message, thus all nodes have the same probability to receive a message. The use of uniform traffic allows us to simulate an unpredictable network load as well as unpredictable used paths. One thousand warm-up messages are generated at the beginning of each test. This represents a 5% of total test time.

Although those solutions can be used for long messages, for the sake of evaluation we consider only single-flit messages. Later, we perform the scalability analysis in an 8×8 2D-mesh configuration with support for up to 64 domains. All configurations include 96-bit-width links. At network edges we statically allocate (16 for a 4×4 network) 1-cycle time slots per node. In a 4×4 topology, one time slot is equal to 1/16 of total bandwidth, resulting to a 6.25% of bandwidth.

6.2.2 Performance Evaluation for IOB Router Architectures

Figure 6.28 shows achieved results for the different strategies. DFCNoC is implemented with the XY and SR routing algorithm. The Token strategy uses XY and SR as well, but then we explored the same philosophy with networks of different latencies (Token x STAGES) and with custom 2D-mesh topologies (Token PERIOD x). These latter are detailed in Appendix A. We can observe that DCFNoC XY and Token with period 16 achieve the best zero-load latency. The XY routing algorithm is already a good option for DCFNoC, since it enables to equalize all I/O paths to the latency of the network diameter, while the SR routing algorithm implies a longer latency. For Token, the routing algorithm is not relevant, as long as it implements minimal-path routing. Its results are not outstanding, since a 2D mesh with 2-cycle hops can natively support only 4 domains, and a large number of delays is required to make 16 domains conflict-free. One could think of decreasing the amount of delays required by Token by implementing links of longer latency (i.e., by spreading delays all around adding router stages), but apart from the way the individual I/O paths are affected, the aggregated network performance is not significantly affected. A better approach for Token consists of customizing the 2D-mesh topology, so that the supported number of domains by construction is larger than 4, for instance 8 or 16. This leads to a progressive reduction of the number of delays, up to the 16-domain solution that is completely delay-free. In this case, the topology then starts looking like an augmented ring (i.e., a ring with bypass links), the latency balance is positive, and the period 16 Token solution matches DCFNoC XY latency. For the sake of comparison, a pure ring topology is also reported, together with the best state-of-the-art solution for low-latency TDM NoCs (called Token-PhaseNoC [2] in the picture). The picture clearly highlights the significant latency savings of the proposed TDM NoCs over state-of-the-art. Finally, note that a No-Delay configuration with standard routing strategy (COUPLES XY) is the only one that has a degraded saturation point. Instead, our customized No-Delay configuration (DIAGONAL) restores the ideal saturation point with improved latency as well, and no delay insertion.

Figure 6.29 shows network end-to-end latencies for all solutions. As shown, DCFNoC XY gets always the same latency, thus average latency coincides with the maximum and minimum ones. COUPLES XY achieves the best network latency but, as shown in Figure 6.28, is the only one that has a degraded saturation point. The Period 16 and

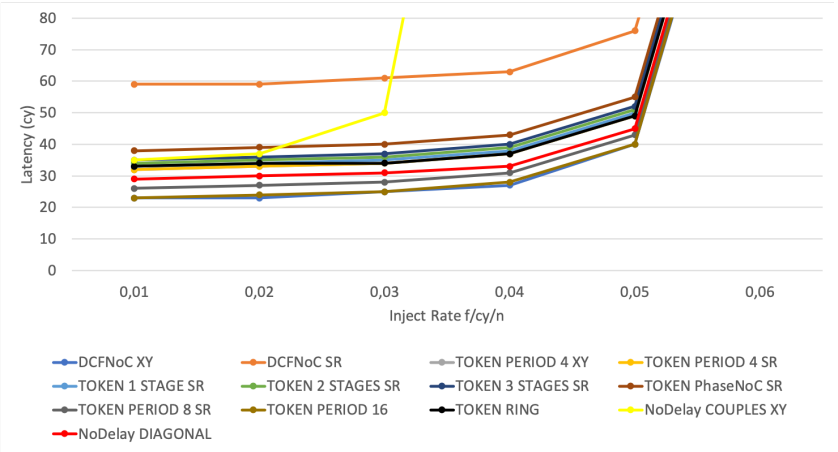


FIGURE 6.28: Load latency comparison for IOB router architectures in a 4×4 network.

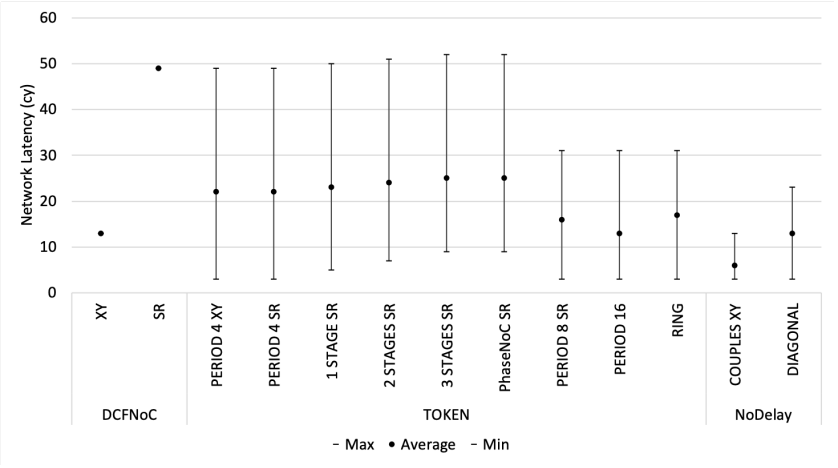


FIGURE 6.29: End-to-End latency comparison for IOB router architectures in a 4×4 network.

DIAGONAL configurations have the second best average network latency. Additionally, Period 16 average is better than the pure ring due to the use of two interconnected links with absence of delays. On the other hand PhaseNoC solution have large differences between the minimum and maximum latencies. Regarding performance guarantees, configurations with lower maximum latency like DCFNoC XY are the ones that provide better latency guarantees.

In order to investigate the performance-cost trade-off of the proposed TDM NoC solutions, we report in Figure 6.30 the implementation costs in terms of number of added delays (in orange) and number of router ports (in blue). Since we are targeting TDM routers that are conflict-free by construction, router I/O ports can be reduced to simple retiming stages, therefore they have the same complexity of added propagation delays:

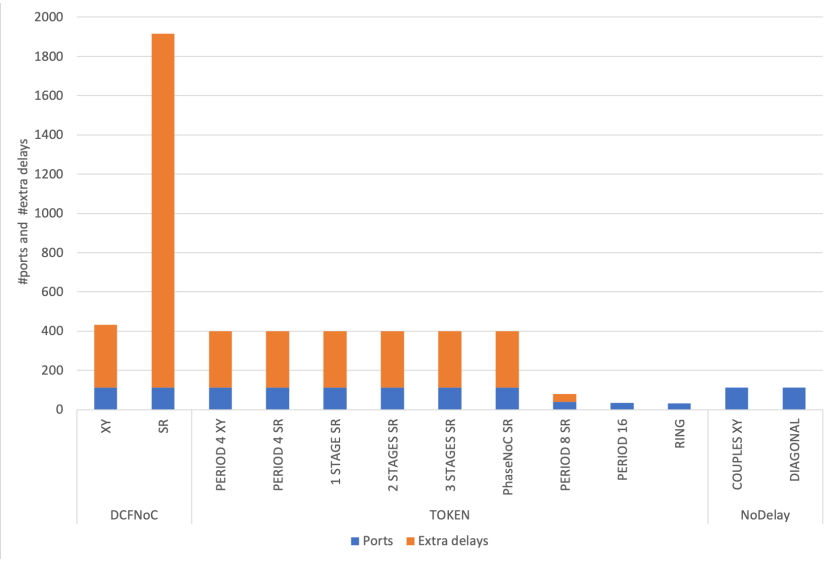


FIGURE 6.30: Cost comparison for IOB router architectures in terms of ports and additional delays in a 4×4 network.

the two major contributions to area can thus be summed up together. The remaining components of the router do not count much, since there is no conventional router allocation phase: there is either a single flit in each router per cycle or there are multiple flits heading to different non-overlapping output ports. Results indicate that DCFNoC XY pays a significant buffering cost to equalize the latency of each I/O path. At the same time, DCFNoC with SR routing is clearly unaffordable. Several Token configurations have almost the same implementation cost of DCFNoC. However, customizing the topology enables the Token philosophy to unfold its benefits, not only in terms of latency (Figure 6.28), but also in terms of area overhead: see the Period 8 and the Period 16 design points in Figure 6.30. Interestingly, Period 16 is delay-less, therefore only the contribution of router I/O ports is accounted for. The same delay freedom is achieved by the No-Delay configurations, although they operate on the 2D-mesh as a whole, thus they are ultimately more costly. Finally, the picture clearly represents the significant cost savings of the proposed TDM NoCs over state-of-the-art.

6.2.3 Performance Evaluation for OB Router Architectures

Figure 6.31 shows load curves of most competitive solutions for OB router architectures. For comparison purposes we also show DCFNoC SR, Token Period 4 XY and Token Period 4 SR to compare the gains with better approaches of both DCFNoC and Token

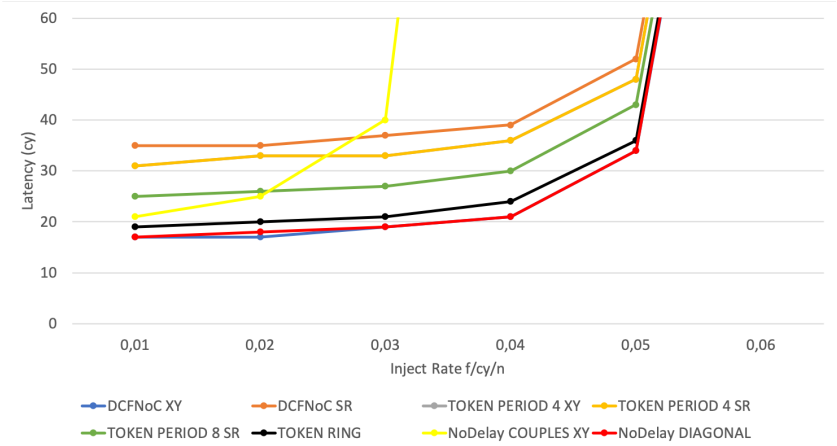


FIGURE 6.31: Load latency comparison for OB router architectures in a 4×4 network.

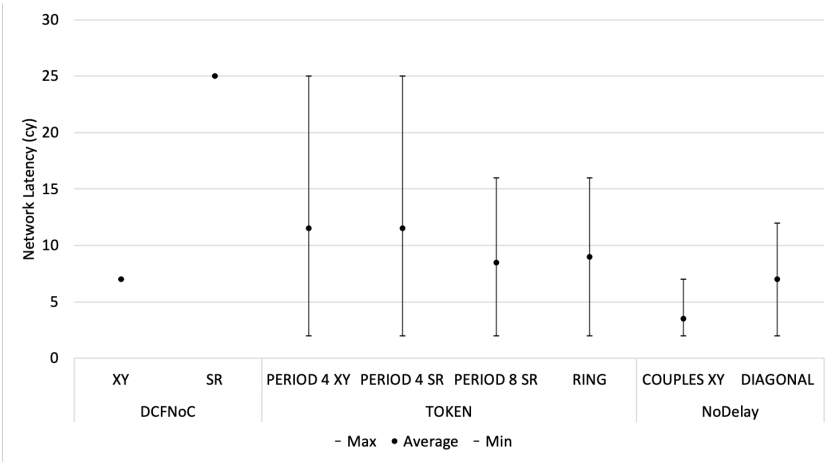


FIGURE 6.32: End-to-End latency comparison for OB router architectures in a 4×4 network.

strategies. Note that a pure ring using OB router architecture in a 4×4 network is equal to a Period 16 solution. Therefore, Period 8 configuration implements two interconnected links. For Token family, the routing algorithm is not relevant, as long as it implements minimal-path routing. The results of token family are not outstanding, since a 2D mesh with 1-cycle hop can natively support only 2 domains, and a large number of delays is required to make 16 delays conflict-free. DCFNoC XY still having the best zero-load latency while DIAGONAL is able to match its strong results. On the other side, the COUPLES XY strategy still suffering from degradation at saturation point.

Network end-to-end latency results for OB router architectures are shown in Figure 6.32. The No-Delay configuration with standard routing strategy (COUPLES XY) achieves the best average latency, however, suffers from degradation at saturation point due to longer TDM period than other solutions. Instead, DCFNoC XY and DIAGONAL solutions are

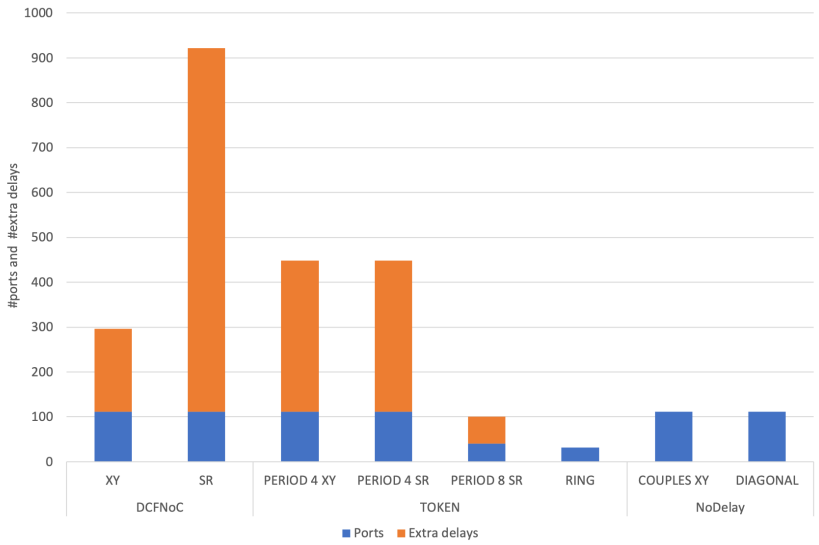


FIGURE 6.33: Cost comparison for OB router architectures in terms of ports and additional delays in a 4×4 network.

able to get the best network latency thanks to an optimal TDM period being equal to the number of network nodes. Period 8 and ring approaches suffer from possible long delays depending on the position of the source and destination node. On the other hand, DCFNoC SR, Token Period 4 XY and Token Period 4 SR require large number of delays to make 16 delays conflict-free. The XY routing algorithm in DCFNoC strategy seems to be the best option to achieve the best guaranteed latency.

Figure 6.33 shows the implementation cost in terms of number of added delays (in orange) and number of router ports (in blue). Results indicate that Token solutions with delays increase buffering cost by 17% to make 16 delays conflict-free. Instead, DCFNoC solutions get a substantial improvement in its results due to reduce hop cost by 50%. In particular, DCFNoC reduces the implementation costs by 43% and 55% for XY and SR routing algorithm, respectively.

6.2.4 Scalability Analysis

In order to evaluate configurations for large network size we provide NoC solutions following the same principles as the ones used for 4×4 network size. In this case, we use IOB router architectures in 8×8 networks. Note that Token solutions with custom 2D-mesh topologies as well as for large network size are provided in Appendix A.

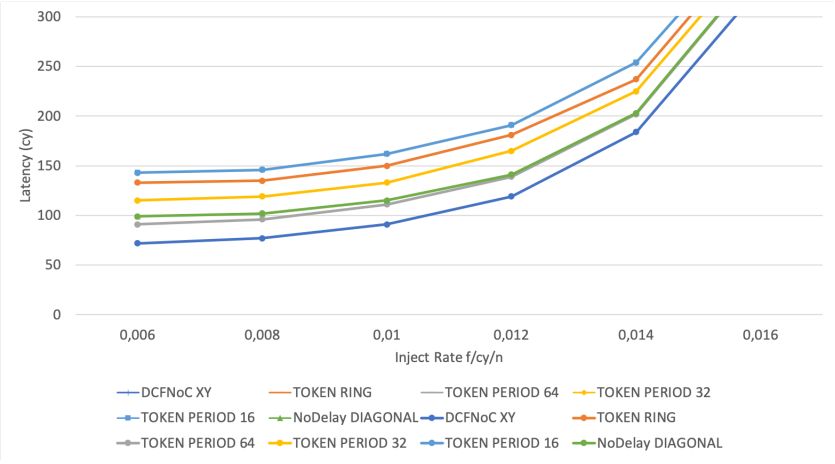


FIGURE 6.34: Load latency comparison for IOB router architectures in a 8×8 network.

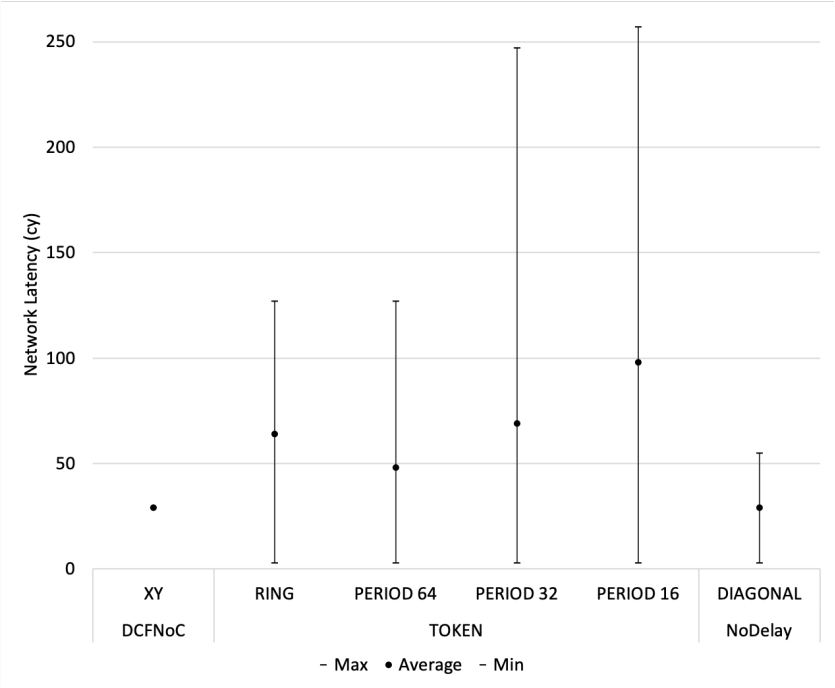


FIGURE 6.35: End-to-End latency comparison for IOB router architectures in a 8×8 network.

Figure 6.34 shows load curves for IOB router architectures in a 8×8 network. DCFNoC XY get the best zero-load latency for large NoC size followed by Token Period 64 and DIAGONAL configurations. Period 64 configuration is a good performance-cost an scalable solution. On the other side, RING suffers from performance degradation for large network size.

Figure 6.35 shows network latency results for a 8×8 network. Token configurations implementing high amount of delays suffers from longer load latency. RING and Period X

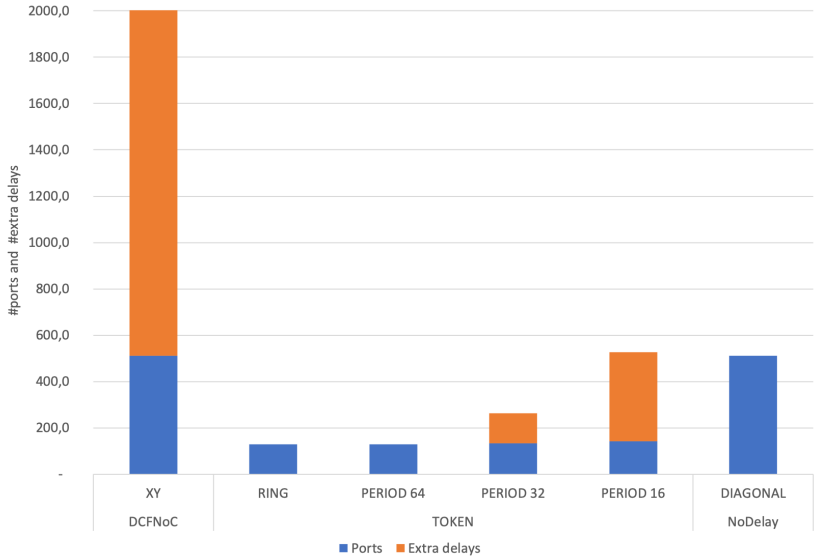


FIGURE 6.36: Cost comparison for IOB router architectures in terms of ports and additional delays in a 8×8 network.

approaches suffer from possible long delays depending on the position of the source and destination node. On the other hand, DIAGONAL and specially DCFNoC XY configurations achieves better guaranteed latency. The average network latency for DCFNoC and DIAGONAL configurations increases 2.23×, leading to the best performance results. For Period 64 approach and RING increases 3.69× and 3.76× respectively.

Figure 6.36 shows the implementation cost for a 8×8 network. Note that in terms of scalability DCFNoC router output port complexity increases as it implements more delays. For Period 32 and Period 16 large number of delays are needed to make 64 delays conflict-free. For the rest of approaches delays are implemented at scheduling level. Therefore, the implementation cost for DCFNoC and DIAGONAL configurations increases 4.63× and 4.57× respectively. For the Period 64 approach and RING increases 3.82× and 4× respectively.

Regarding DCFNoC scalability using IOB/OB architectures, DCFNoC achieves the best zero-load latency for large NoC size, however the IOB architecture is not able to squeeze the best of this methodology, as doubles the number of additional delays to implement compared to IOB architecture. In fact, DCFNoC using OB routers is able to improve the network latency being reduced to 15 cycles, half compared to IOB architecture. Moreover, the number of additional delays are also halved. Therefore, DCFNoC achieves

the best attainable maximum network latency (guaranteed latency) of all conflict-free solutions using OB architecture.

6.2.5 Area Overhead And Frequency

In order to implement different routing algorithms in a simply and efficient way, we use the Logic Based Distributed Routing (LBDR) algorithm [72]. LBDR is a distributed and implementation-efficient methodology adapted to irregular networks. By using LBDR mechanism we are able to implement XY and SR routing algorithms in a simple and efficient way. This technique provides a good scalability as LBDR complexity just depends on the number of router I/O ports and not on the network size [73].

Maximum operating frequency and area utilization have been obtained using Cadence RC Compiler and the 45-nm Nangate library [60]. Regarding area overhead results, we already reported topology-level ports and added delays for all router proposals. In this section we focus on the most competitive routers.

Figure 6.37 shows area overheads for IOB router architectures when targeting high frequency. Those routers are DCFNoC, Token Period 4 SR, Period 16 and DIAGONAL in a 4×4 network and also DCFNoC XY in a 8×8 network. The 4×4 mesh Period 16 data router uses 82.57% less area than the DCFNoC XY one and 17.95% less area than the DIAGONAL for a 4×4 mesh. DCFNoC XY router implements a crossbar interconnect as well as output delay registers. On the other hand, the DIAGONAL router is delay-less but implements more output ports as well as routing logic to compute the output port. Due to the small number of ports the Period 16 router is the lightest router for a 4×4 mesh implementation with a total area of $2,593\ mm^2$.

Figure 6.37 also shows total area of DCFNoC XY router implementation for a 8×8 mesh, as it achieves the best zero-load latency for a 8×8 network, with only 12% more area overhead compared with the 4×4 implementation. The DCFNoC XY router area overheads may be reduced using the OB architecture.

We have also analyzed the maximum attainable clock frequency of the different routers. Figure 6.38 shows that the simpler Period 16 router design achieves a significant boost in clock frequency by improving Token-based Period 4 router's one by 25%. The critical

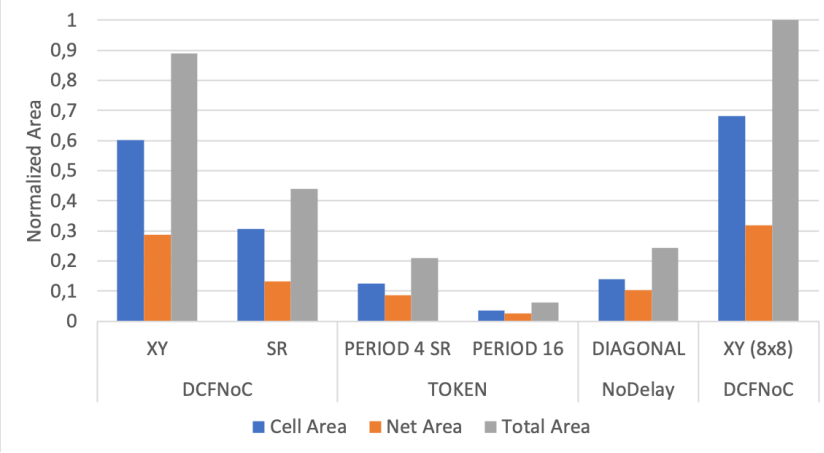


FIGURE 6.37: Area overhead for IOB router architectures in a 4×4 network and DCFNoC in a 8×8 network.

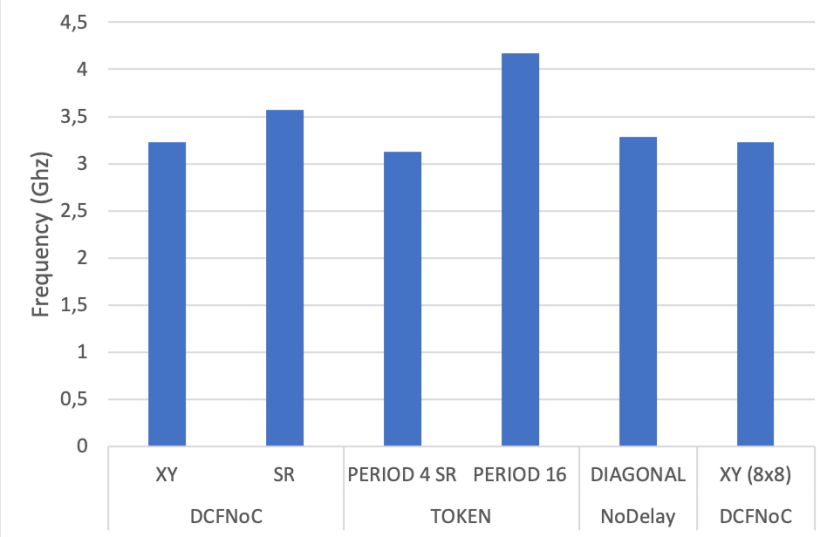


FIGURE 6.38: Maximum attainable clock frequency for IOB router architectures in a 4×4 network and DCFNoC in a 8×8 network.

path of the Token-based Period 4 router limits clock frequency to 3.125 GHz. However, the Period 16 router exhibits a critical path of 240 ps leading to a maximum clock frequency of 4.17 GHz.

For the DCFNoC XY router, even with such best low zero-load latency is able to reach a clock frequency above the average of most competitive routers. DCFNoC XY clock frequency reaches up to 3.22 GHz for both 4×4 and 8×8 mesh implementations.

6.3 Summary

In this chapter, we have characterized the cost-performance trade-offs of a wide range of options to build a low-latency and low-cost TDM NoC which will be beneficial in current and future safety-critical real-time systems. All solutions analysed in this chapter are inspired by the observation of the channel dependency graph, and significant improvements are achieved over state-of-the-art: up to 20× lower area and 40% latency savings for Period 16 and DCFNoC XY, respectively.

6.4 Acknowledgment

The work in this chapter has been done during my internship at Università degli Studi di Ferrara, Ferrara, Italy. My tutor at University of Ferrara, Davide Bertozzi, guided me to develop all the work done, meanwhile, we had a great time there.

Chapter 7

Conclusions

In this chapter we summarize the main conclusions drawn in this thesis. First, we summarize the specific contributions of the proposals, then we follow a discussion about future work, and finally we provide an enumeration of the scientific publications related with this dissertation.

7.1 Contributions

MPSoCs have been recently introduced in new environments like avionics or automotive. These new domains introduce challenging requirements, such as time predictability and performance isolation, that demand for alternative NoC designs. Future safety-critical real-time systems will need processor designs able to provide performance guarantees without renouncing to peak throughput numbers.

In this thesis, we target a time-predictable NoC design paradigm able to provide timing guarantees while enabling competitive performance. To address the communication contention problem we rely on TDM scheduling to provide configurable bandwidth reservation, guaranteed latency and throughput. We aim to find TDM schedules close to the theoretical lower bound without using a computationally demanding offline process. Next, we integrate our time-predictable NoC into a manycore design to test performance guarantees in non congested/saturated conditions. Then, we test the strong temporal isolation between communication flows needed in safety-critical MPSoCs. Finally, in order to provide peak throughput numbers while preserving strict real-time guarantees we propose a dynamic scheduler.

As a first contribution, a new time-predictable NoC design paradigm based on the CDGs theory has been proposed. DCFNoC guarantees by design the avoidance of contention within the NoC. The proposed approach improves over state-of-the-art TDM proposals by achieving scheduling periods that almost match the theoretical lower bound. While traditional approaches have difficulties to find schedules for large networks, DCFNoC is able to find conflict-free scenarios in arbitrarily large NoC sizes without degrading the quality of the achieved guarantees. Finally, we have also shown the feasibility of the proposed approach by implementing a high-speed router design with very small area needs.

As a second contribution, DCFNoC has been integrated in a tile-based manycore system and adapted to its memory hierarchy designing a new network interface module. Experimental results show that DCFNoC guarantees time predictability avoiding network interference among multiple running applications. DCFNoC always guarantees performance and also improves wormhole performance in a 4×4 setting by a factor of 3.7×

when interference traffic is injected. For a 8×8 network differences are even larger. In addition, DCFNoC obtains a total area saving of 10.79% over a standard wormhole implementation.

As a third contribution, a dynamic and distributed scheduler has been presented to improve peak performance. The proposed scheduler builds on top of DCFNoC to exploit its unique features and is able to achieve peak performance that is very close to a wormhole-based NoC design. At the same time, this design keeps the real-time guarantees of DCFNoC. Experimental results show that the proposed scheduler achieves an overall throughput improvement of 6.9× and 14.4× over a baseline DCFNoC for 16 and 64-node meshes, respectively. When compared against a standard wormhole router 95% of its network throughput is preserved.

Finally, as a fourth contribution a study of the best TDM NoC approaches is provided. The cost-performance trade-off of a wide range of solutions has been characterized to achieve TDM-based NoC with low-latency and low-cost. All solutions are inspired by the observation of the channel dependency graph, and in some cases significant improvements are achieved over state-of-the-art, up to 20× lower area and 40% latency savings.

As a result, all the purposes outlined in this thesis have been reached successfully by these contributions.

7.2 Future Directions

In this Section we highlight possible future research directions coming from this dissertation. The following is a list of possible extensions:

- **Define a framework to design TDM-based real-time NoCs.** This dissertation has been focused to implement different TDM NoC solutions. All of these solutions are based on the CDGs theory to identify potential conflicts within the NoC. In Chapter 3, a TDM-based NoC design paradigm has been presented and formalized where conflicts are avoided by serializing message transmissions. In this sense, a new framework based on the CDGs theory to identify potential conflicts within the NoC can be properly defined and will serve to tailor NoC designs to specific applications.

- **Implement a predictable snoopy coherence protocol exploiting DCFNoC properties.** Snoopy protocols have a non-negligible complexity when implemented in 2D meshes due to the need for broadcast support. DCFNoC have broadcast native support. DCFNoC broadcast support is used in Chapter 5 for notification phase between nodes. DCFNoC makes a mesh network to behave like a bus, thus being easier to implement and validate a time predictable cache coherence protocol. DCFNoC broadcast benefit can be exploited to implement a time predictable snoopy coherence protocol. How to design time predictable cache coherence protocols for safety-critical real-time systems is an open research problem.

- **Implement a time predictable and partitioned manycore solution by using hierarchical DCFNoC networks.** In future MPSoCs, as the number of implemented IP cores increases, efficient communication among them and with off-chip resources becomes key to achieve the intended performance scalability. In fact, the TDM-based NoC architectures should be customized to enable full reconfigurability. At run-time, the number of running partitions/domains on the NoC can range from 1 to the number of processing cores, and the number can change over time depending on instantaneous resource requirements of each real-time application. Therefore, the chosen scheme should enable such dynamic reconfiguration of the number of running partitions, while still delivering low latency and partition isolation. In Chapter 6 we tackle the DCFNoC implementation with LBDR mechanism that provides partition support as well as partition reconfiguration mechanism, while strict timing predictability and partition isolation property is kept. In this sense, a partitioned manycore solution should implement one DCFNoC network per partition and another DCFNoC network at a higher hierarchical level to provide main memory access guarantees. How to implement such architectures in an efficient manner is an open problem.

7.3 Publications

The following papers related with the main contributions of this dissertation has been submitted and accepted for publication in different international journals and conferences with peer review.

Journals:

- T. Picornell, J. Flich, J. Duato, and C. Hernández. hp-DCFNoC: High Performance Distributed Dynamic TDM Scheduler based on DCFNoC Theory. *IEEE Access*, volume 8, pages 194836-194849, 2020.
- T. Picornell, J. Flich, C. Hernández, and J. Duato. Enforcing Predictability of manycores with DCFNoC. *IEEE Transactions on Computers* (TC), volume 70, issue 2, pages 270-283, 2021.

Conferences:

- T. Picornell, J. Flich, C. Hernández, and J. Duato. DCFNoC: A Delayed Conflict-Free Time Division Multiplexing Network on Chip. In *Proceedings of the 56th Annual Design Automation Conference 2019* (DAC), pages 1-6, Las Vegas, NV, USA, 2019.

In addition, other related papers have been published in international summer schools and domestic conferences:

- T. Picornell, J. Flich, C. Hernández, and J. Duato. DCFNoC: A new time-predictable NoC design paradigm. In *Proceedings of the 15th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES), pages 139-142, Fiuggi, Italy, 2019.
- T. Picornell, J. Flich, R. Tornero, and JM. Martínez. Router Design for Bandwidth Reservation Guarantees. In *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES), pages 291-194, Fiuggi, Italy, 2017.
- T. Picornell, J. Flich, R. Tornero, and JM. Martínez. Arquitectura de Red con Reserva de Anchos de Banda para Sistemas Heterogéneos Basados en FPGAs. In *Actas de las XXVIII Jornadas de Paralelismo* (JP), Málaga, Spain, 2017.

All works listed above are exclusively related with this thesis. The specific contributions of the Ph.D. candidate reside mostly in the design and implementation of the proposed designs, as well as the execution of the performed experiments, the analysis and discussion of the results, the writing of the paper drafts describing the work, and the presentation of the papers in the conferences. Along these processes, the co-authors have repeatedly provided useful hints and advices, which the Ph.D. candidate has then applied to make the work evolve into its final version.

Appendix A

Extending Token Propagation-Based Family

This appendix extends from token propagation-based family presented in Chapter 6. These solutions consists of customizing the 2D-mesh topology to support larger number of domains by construction as well as for large network size. Additional definitions are provided to generalize this philosophy.

A.1 Supporting Specific Topologies with Unidirectional Links

Definition A1. *An unidirectional ring of* 4 *routers has a period of* 8 *cycles using IOB routers.*

As an alternative, in order to support higher number of domains without adding delays, remove links to build a topology with combined and overlapped rings. Therefore, avoiding the addition of delays.

Let us consider a ring of 4 IOB routers in which an initial router receives a token in $t = x$, see Figure A.1. Since every propagation hop takes two cycles and the token takes four hops, the token arrives back in $t = x + 8$. Therefore, the relative latency is 8 cycles.

Definition A2. *If we combine unidirectional rings of* 4 *routers in a 2D mesh we preserve an optimal number of domains equal to* 8*.*

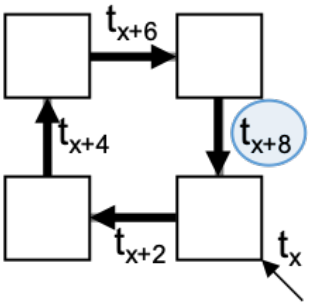


FIGURE A.1: Unidirectional ring of 4 routers with a period of 8 cycles using IOB routers.

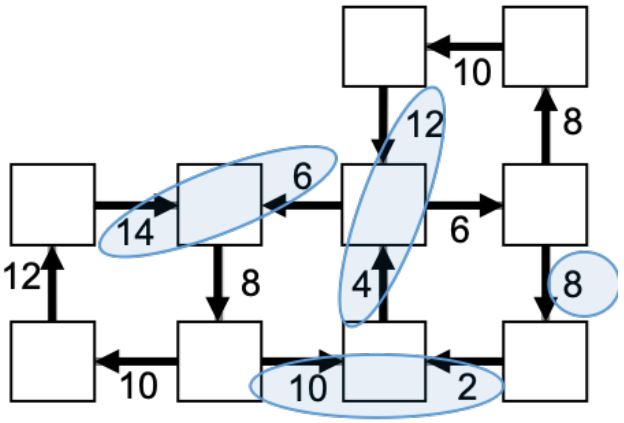


FIGURE A.2: Combining multiple unidirectional rings of 4 preserving a relative latency of 8 cycles in every converging point using IOB routers.

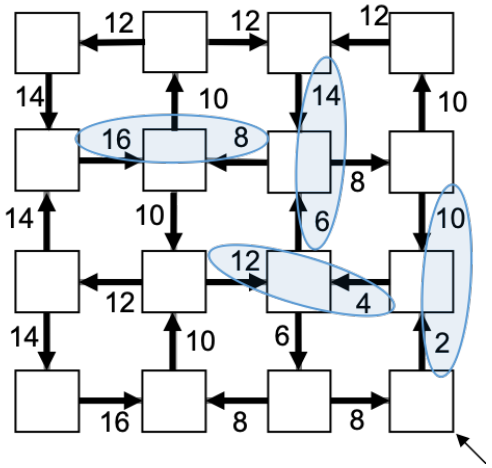


FIGURE A.3: Optimal number of domains equal to 8 in a 4×4 mesh using IOB routers by only combining unidirectional rings of 4 routers. The token is injected from the bottom right-most node.

Let us consider the following combination of rings, see Figure A.2. Numbers represent token propagation latency in cycles. As we can see, by following token propagation to connected rings, we can preserve a relative latency of 8 cycles in every converging point.

We exploit this property and present a new topology with a full combination of unidirectional rings of 4 routers in a 2D mesh, see Figure A.3. This topology is a minimum-cost and delay-free solution in a 2D mesh.

Definition A3. *Given two combined and overlapped rings with period* 8*, the minimum-cost solution to achieve a token ID synchronization is by adding a* $D-8$ *number of delays on the shared link. As a result, a token ID propagation process is synchronized.*

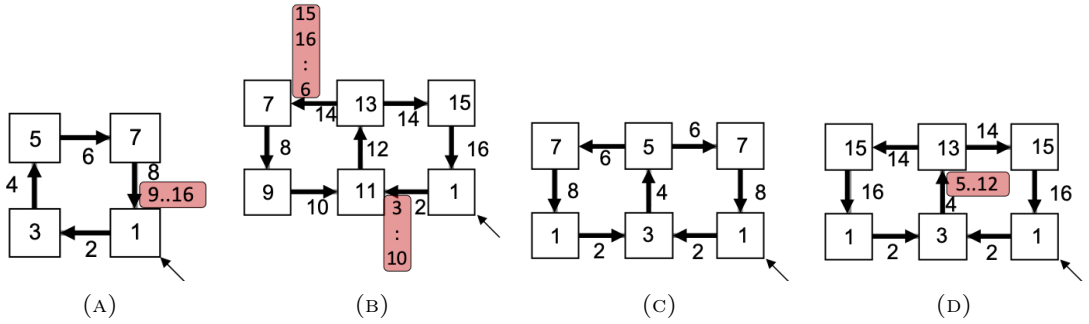


FIGURE A.4: Breaking unidirectional rings by additional delays to support more domains using IOB routers. Additional delays are represented by red rectangles.

Let us consider the following situation shown in Figure A.4a with an unidirectional ring of 4 IOB routers. In order to enlarge the period at converging points, we add a $D - 8$ delay either in shared or non-shared links.

With $D > 8$ domains the ways to synchronize are:

Ⓐ Add delays on non-shared links (arbitrarily) have a cost of two delay points, see Figure A.4b. If you change the position you get the same result.

Ⓑ Add delays on a shared link have a cost of one delay point, see Figure A.4d. Therefore, is the minimum-cost solution.

A.2 Algorithm To Insert Delays on Overlapped Rings

The algorithm is shown in Algorithm A.1. For the sake of explanation, when two overlapped rings or squares are combined, we call this an eight shape formation (i.e., see Figure A.4c). First, in lines 7-12 all links of all rings are labeled as an unsolved. Then, for every eight shape formation (lines 13-24) the direction is obtained to get the shared link between two rings in this eight shape formation. Additional delays are assigned to the shared link and all links included in this eight shape are labeled as solved. Once all eight shapes are solved, rings are inspected to be solved (lines 25-33). Notice that rings may be already solved in an eight shape. When all eight shapes are solved there is only one remaining ring to solve and delays have to be added in one of the unsolved links that does not appear into a previous solved eight shape. An important rule is that a ring has to be broken only once by adding delays.

```
 1: function insert_delays(I,D,P)
 2:  domains = D
 3:  period = P
 4:  eight e
 5:  ring r
 6:  link l
 7:  for every ring in I (r)
 8:     for every link in r (l)
 9:         l.stat = unsolved
10:         l.delays = 0
11:     endfor
12:  endfor
13:  for every eight in I (e)
14:     if (e.direction == vertical)
15:         l = get_vertical_shared_link(e)
16:         l.delays = D-P
17:     end else
18:         l = get_horizontal_shared_link(e)
19:         l.delays = D-P
20:     end
21:     for every link in e (l)
22:         l.stat = solved
23:     endfor
24:  endfor
25:  for every ring in I (r)
26:     for every link in r (l)
27:         if (l.stat == unsolved)
28:             l.delays = D-P
29:             l.stat = solved
30:             break
31:         end
32:     endfor
33:  endfor
34: end function
```

Alg. A.1: Algorithm to insert delays

As a result we provide a full combination of unidirectional rings of 4 IOB routers with a period of 16 by adding delays, see Figure A.5. This is a Period 8 solution which have a natural period of 8 cycles and uses additional delays (8 cycles in every red square) to enlarge the period to 16 cycles, and hence support 16 domains. As we can see, Algorithm A.1 is applied to insert delays.

To define the TDM slot assignment we need to take into account the token propagation shown in Figure A.5. In this particular case even nodes and odd nodes do not coincide as previous solutions. In this case, we assign pointers in the following manner, see Figure A.6.

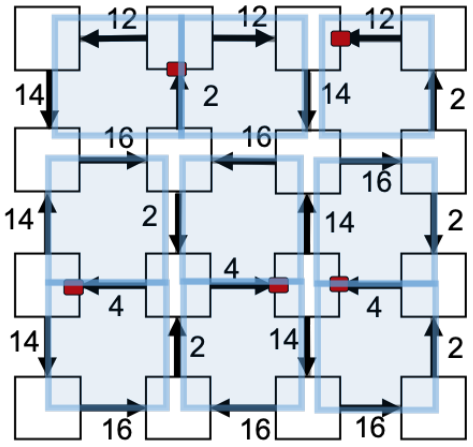


FIGURE A.5: Period 8 solution in a 4×4 mesh by combining unidirectional rings of 4 IOB routers and adding delays (8 cycles in every red square) to enlarge the period to 16 cycles. Algorithm A.1 is applied to insert delays. Additional delays are represented by red squares.

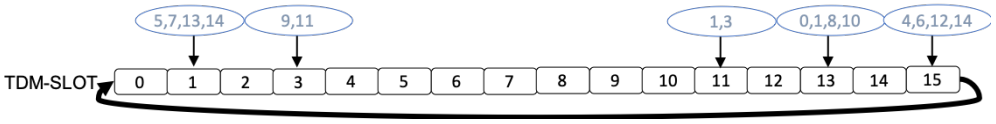


FIGURE A.6: TDM slot wheel assignment to manage message injection in a 4×4 mesh by combining unidirectional rings of 4 IOB routers with support for 16 domains.

A.3 Extending to 16 Domains

Definition A4. *An unidirectional ring of* 8 *routers has a period of* 16 *cycles. If we combine unidirectional rings of* 8 *IOB routers in a 2D mesh we preserve an optimal number of domains equal to* 16.

Figure A.7 shows an unidirectional ring of 8 routers with a period of 16 cycles.

As a result we present a new delay-free topology with a full combination of unidirectional rings of 8 IOB routers in a 2D mesh, see Figure A.8. This a Period 16 solution which has a natural period of 16 cycles, and hence supports 16 domains. As we can see, relative latencies are always 16 cycles at every converging point.

To define the TDM slot assignment we follow the same methodology as the previous solution taking into account the token propagation shown in Figure A.8. In this case, we assign pointers in the following manner, see Figure A.9.

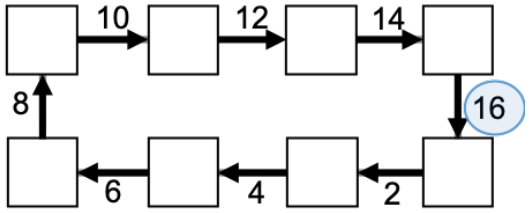


FIGURE A.7: Unidirectional ring of 8 routers with a period of 16 cycles.

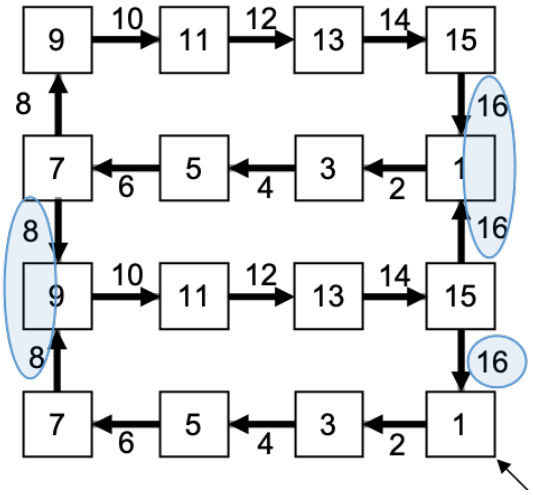


FIGURE A.8: Period 16 topology in a 4×4 mesh by only combining unidirectional rings of 8 routers using IOB routers. This is a delay-free solution.

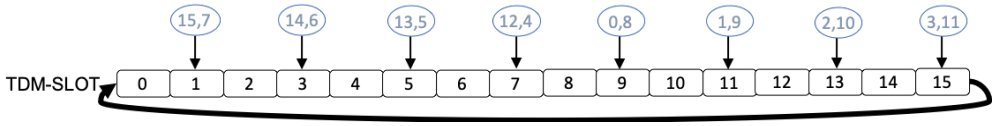


FIGURE A.9: TDM slot wheel assignment to manage message injection in a 4×4 mesh by combining unidirectional rings of 8 routers with support for 16 domains.

A.4 Extending to Larger Networks

Following the last definitions we can get solutions for large network size such as 8×8 mesh using IOB routers. Figure A.10 shows a new topology with a combination of unidirectional rings of 16 routers with relative latencies of 32 cycles. This is a Period 32 solution which has a natural period of 32 cycles and uses additional delays (32 cycles in every red square) to enlarge the period to 64 cycles, and hence support for 64 domains. In order to enlarge the period we extend every ring using additional delays of 32 cycles following Algorithm A.1.

A delay-free solution merges from Period 32 by making rings longer. Figure A.11 show a delay-free solution with a combination of unidirectional rings of 32 routers with relative latencies of 64 cycles. This a Period 64 solution which has a natural period of 64 cycles, and hence support for 64 domains.

On the other hand, a different solution emerges from Period 32 by using shorter rings and using more delays. Figure A.12 shows a solution using a combination of unidirectional

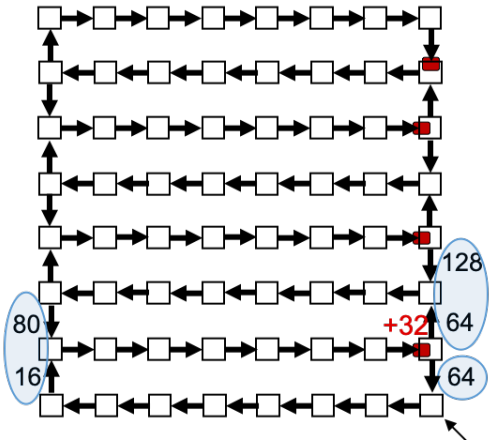


FIGURE A.10: Period 32 topology in a 8×8 mesh by combining unidirectional rings of 16 routers and adding delays (32 cycles in every red square) to enlarge the period to 64 cycles. Implemented delays are 32 cycles at every red point. This is an extension of Period 16 for a 4×4 mesh

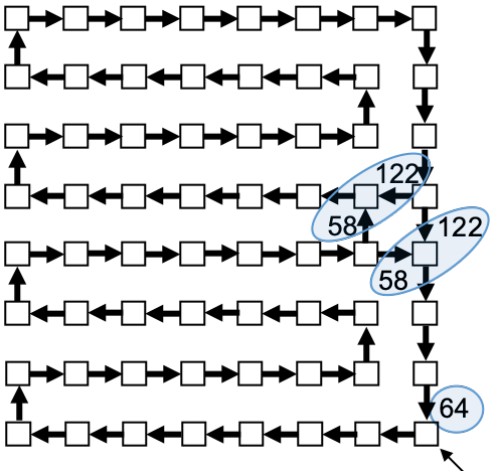


FIGURE A.11: Period 64 topology in a 8×8 mesh by combining unidirectional rings of 32 routers. This is a delay-free solution.

rings of 8 routers with relative latencies of 16 cycles. This a Period 16 solution in a 8×8 mesh which has a natural period of 16 cycles and uses additional delays (48 cycles in every red square) to enlarge the period to 64 cycles, and hence support for 64 domains.

As a result of this family of solutions we presented the following solutions:

Solutions for a 4×4 network:

- **Period** 4: Natural period of 4 cycles and uses additional delays (12 cycles in every red square) to enlarge the period to 16 cycles, and hence support for 16 domains.

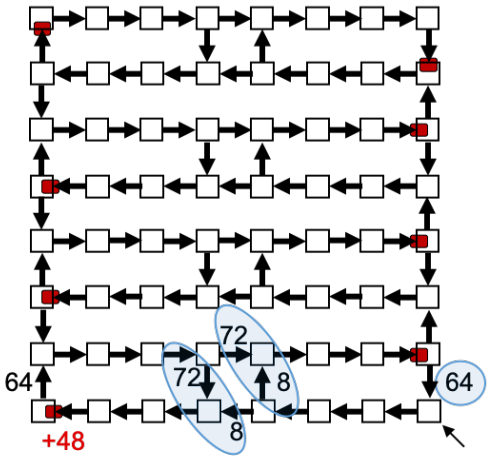
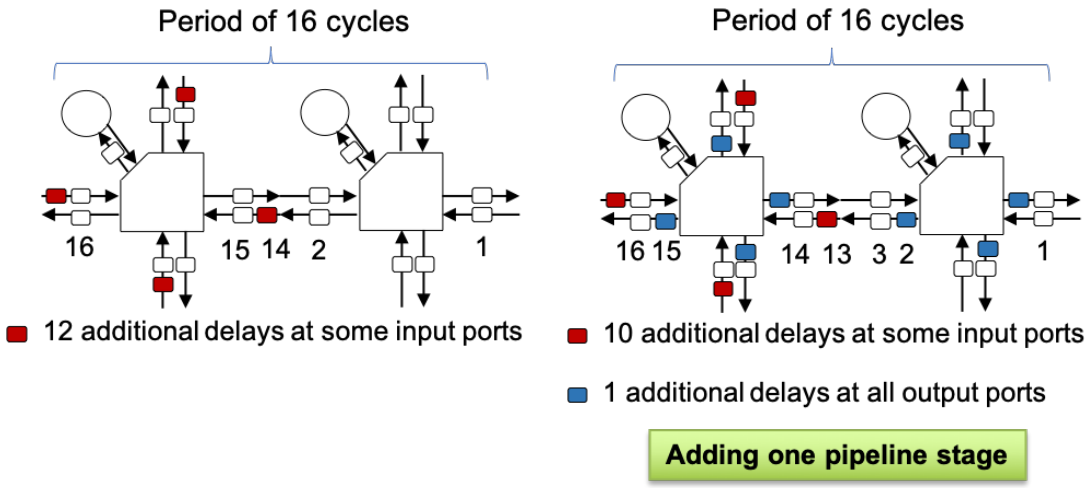


FIGURE A.12: Period 16 topology in a 8×8 mesh by combining unidirectional rings of 16 routers and adding delays enlarge the period to 64 cycles. Implemented delays are 48 cycles at every red point.

- **Period** 8: Natural period of 8 cycles and uses additional delays (8 cycles in every red square) to enlarge the period to 16 cycles, and hence support for 16 domains.
- **Period** 16: Delay-free solution with natural period of 16 cycles, and hence support for 16 domains.

Solutions for a 8×8 network:

- **Period** 32: Natural period of 32 cycles and uses additional delays (32 cycles in every red square) to enlarge the period to 64 cycles, and hence support for 64 domains.
- **Period** 64: Delay-free solution with natural period of 64 cycles, and hence support for 64 domains.
- **Period** 16: Natural period of 16 cycles and uses additional delays (48 cycles in every red square) to enlarge the period to 64 cycles, and hence support for 64 domains.



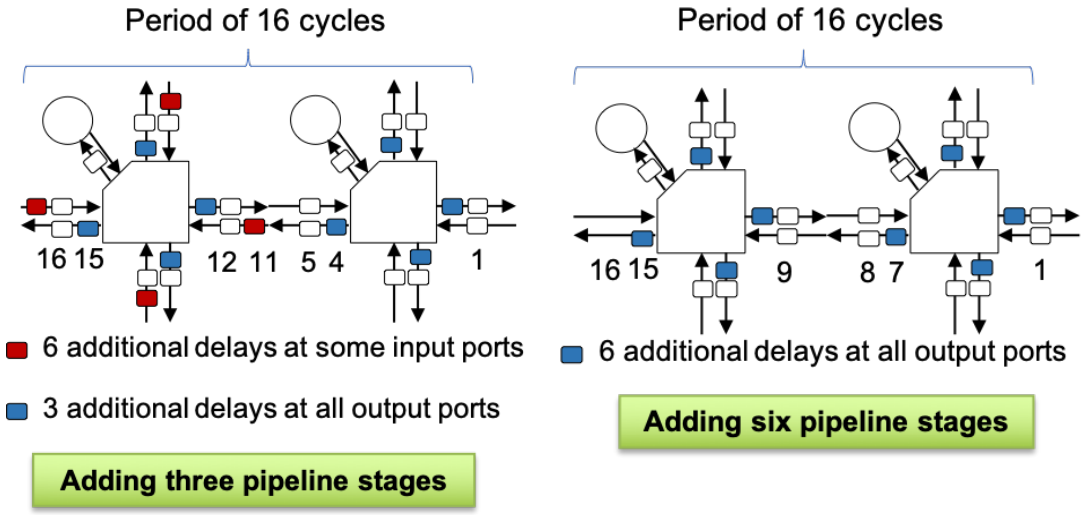
(A) (B)

FIGURE A.13: Period between two consecutive routers in Period 4 topology (a). In order to spread the added delays, we move them from some input ports to all output ports, and hence adding one pipeline stage (b).

A.5 Tuning Router Pipeline Stages to Make Latency Hotspots Lighter

Previous solutions uses additional delays. Now, we analyze the impact of varying the location of those additional delays. This would spread the delays along the network. To do so, we first analyse the period between two consecutive routers in Period 4 topology, see Figure A.13a. Additional delays are located once every period of 16 cycles. In order to spread the delays we can move those delays along the network as Figure A.13b shows. With this method we move the additional delays located at some input ports gradually by adding delays at all output ports. As Figure A.13b shows we are adding one pipeline stage.

By following the same methodology we can study how spreading the additional delays by adding additional pipeline stages affects the network latency. Figure A.14a shows a setup of three additional pipeline stages. We end up by equally spreading the additional delays in pipeline stages, see Figure A.14b. In this setup, every network hop becomes equal in terms of cycles, hence latency hotspots are avoided.



(A) (B)

FIGURE A.14: Gradually moving the added delays by adding two pipeline stages (a). By adding six pipelines stages every hop have the same length (b).

References

[1] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 152–160, May 2012. doi: 10.1109/NOCS.2012.25.

[2] A. Psarras, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos. PhaseNoC: TDM scheduling at the virtual-channel level for efficient network traffic isolation. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1090–1095, March 2015. doi: 10.7873/DATE.2015.0418.

[3] Miguel Gorgues Alonso, José Flich, Meriem Turki, and Davide Bertozzi. A Low-Latency and Flexible TDM NoC for Strong Isolation in Security-Critical Systems. In *13th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2019, Singapore, Singapore, October 1-4, 2019*, pages 149–156. IEEE, 2019. doi: 10.1109/MCSoC.2019.00029. URL `https://doi.org/10.1109/MCSoC.2019.00029`.

[4] Transparency Market Research. Embedded System Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2019 - 2027. `http://www.transparencymarketresearch.com/embedded-system.html`, 2019.

[5] Transparency Market Research. Embedded Automation Computers Market - Global Industry Analysis, Size, Share, Growth, Trends, and Forecast, 2019 - 2027. `https://www.transparencymarketresearch.com/embedded-automation-computers-market.html`, 2019.

[6] CBI. Promising European markets for embedded systems for telemedicine. https://www.cbi.eu/market-information/electronics-electrical-engineering/embedded-systems-telemedicine/, 2017.

[7] GlobeNewswire. Global Embedded System Market (2020 to 2025) - Rapid Adoption of Embedded Systems in Smart Homes Presents Lucrative Opportunities. https://www.globenewswire.com/news-release/2020/03/26/2006966/0/en/Global-Embedded-System-Market-2020-to-2025-Rapid-Adoption-of-Embedded-Systems-in-Smart-H html, 2020.

[8] ISO 26262-1:2018. Road vehicles — Functional safety. URL https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en.

[9] Grand View Research. Automotive Electronic Control Unit Market Size, Share, & Trends Analysis Report By Application, By Propulsion Type, By Capacity, By Vehicle Type, By Region, And Segment Forecasts, 2019 - 2025, 2019. URL https://www.grandviewresearch.com/industry-analysis/automotive-ecu-market.

[10] Embitel. 'ECU' is a Three Letter Answer for all the Innovative Features in Your Car: Know How the Story Unfolded. https://www.embitel.com/blog/embedded-blog/automotive-control-units-development-innovations-mechanical-to-electronics, 2017.

[11] ABB. Trends in embedded systems, 2006. URL https://library.e.abb.com/public/3342a1f0430d9c36c12571930034abe5/09-13%202M620_ENG72dpi.pdf.

[12] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421, Sept 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

[13] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A flit-synchronous network on chip with composable and predictable services. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 250–255, April 2009. doi: 10.1109/DATE.2009.5090666.

[14] R. A. Stefan, A. Molnos, and K. Goossens. daelite: A tdm noc supporting qos, multicast, and fast connection set-up. *IEEE Transactions on Computers*, 63(3):583–594, March 2014. ISSN 0018-9340. doi: 10.1109/TC.2012.117.

[15] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 890–895 Vol.2, Feb 2004. doi: 10.1109/DATE.2004.1269001.

[16] Ni. Issues in designing truly scalable interconnection networks. In *1996 Proceedings ICPP Workshop on Challenges for Parallel Processing*, pages 74–83, 1996. doi: 10.1109/ICPPW.1996.538592.

[17] D. Ludovici, F. Gilabert, S. Medardoni, C. Gómez, M. E. Gómez, P. López, G. N. Gaydadjiev, and D. Bertozzi. Assessing Fat-Tree Topologies for Regular Network-on-Chip Design under Nanoscale Technology Constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, page 562–565, Leuven, BEL, 2009. European Design and Automation Association. ISBN 9783981080155.

[18] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, J.F. Brown, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *Micro, IEEE*, 27:15 – 31, 10 2007. doi: 10.1109/MM.2007.4378780.

[19] Intel Corporation. URL `http://www.intel.com`.

[20] Y. Tamir and H. C. Chi. Symmetric Crossbar Arbiters for VLSI Communication Switches. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):13–27, January 1993. ISSN 1045-9219. doi: 10.1109/71.205650. URL `https://doi.org/10.1109/71.205650`.

[21] R.O. LaMaire and D.N. Serpanos. Two-dimensional round-robin schedulers for packet switches with multiple input queues. *IEEE/ACM Transactions on Networking*, 2(5):471–482, 1994. doi: 10.1109/90.336324.

[22] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010. ISBN 1439837104, 9781439837108.

[23] Dally and Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.1676939.

[24] Charles L. Seitz and Wen-King Su. A Family of Routing and Communication Chips Based on the Mosaic. In *Proceedings of the 1993 Symposium on Research on Integrated Systems*, page 320–337, Cambridge, MA, USA, 1993. MIT Press. ISBN 0262023571.

[25] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[26] M. Karol, M. Hluchyj, and S. Morgan. Input Versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communications*, 35(12):1347–1356, 1987. doi: 10.1109/TCOM.1987.1096719.

[27] W.J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, 1992. doi: 10.1109/71.127260.

[28] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997. ISBN 0818678003.

[29] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4 (4):466–475, April 1993. ISSN 1045-9219. doi: 10.1109/71.219761.

[30] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, Dec 1993. ISSN 1045-9219. doi: 10.1109/71.250114.

[31] Christopher J. Glass and Lionel M. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, page 278–287, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915097. doi: 10.1145/139669.140384. URL `https://doi.org/10.1145/139669.140384`.

[32] Andres Mejia, Jose Flich, and José Duato. On the Potentials of Segment-Based Routing for NoCs. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 594–603. IEEE Computer Society, 2008. doi: 10.1109/ICPP.2008.56. URL `https://doi.org/10.1109/ICPP.2008.56`.

[33] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[34] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards Composable Timing for Real-Time Programs. In *2009 Software Technologies for Future Dependable Distributed Systems*, pages 1–5, 2009. doi: 10.1109/STFSSD.2009.26.

[35] Josef Berwanger, M. Peller, and R. Griessbach. byteflight - A New Protocol for Safety Critical Applications. 2000.

[36] R. Makowitz and C. Temple. Flexray - A communication network for automotive control systems. *2006 IEEE International Workshop on Factory Communication Systems*, pages 207–212, 2006.

[37] Salma Hesham, Jens Rettkowski, Diana Göhringer, and Mohamed A. Abd El Ghany. Survey on Real-Time Network-on-Chip Architectures. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 191–202, Cham, 2015. Springer International Publishing. ISBN 978-3-319-16214-0.

[38] Salma Hesham, Jens Rettkowski, Diana Goehringer, and Mohamed A. Abd El Ghany. Survey on Real-Time Networks-on-Chip. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1500–1517, May 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2623619. URL `https://doi.org/10.1109/TPDS.2016.2623619`.

[39] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. *Guaranteeing the Quality of Services in Networks on Chip*, pages 61–82. Springer US, Boston, MA, 2003. ISBN 978-0-306-48727-9. doi: 10.1007/0-306-48727-6_4. URL `https://doi.org/10.1007/0-306-48727-6_4`.

[40] D. Wiklund and Dake Liu. SoCBUS: switched network on chip for hard real time embedded systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2003. doi: 10.1109/IPDPS.2003.1213180.

[41] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Second International Symposium on Networks-on-Chips, NOCS 2008, 5-6 April 2008, Newcastle University, UK. Proceedings*, pages 161–170, 2008. doi: 10.1109/NOCS.2008.11. URL http://doi.ieeecomputersociety.org/10.1109/NOCS.2008.11.

[42] Chun-Hsien Lu, Kuo-Cheng Chiang, and Pao-Ann Hsiung. Round-based priority arbitration for predictable and reconfigurable Network-on-Chip. In *2009 International Conference on Field-Programmable Technology*, pages 403–406, 2009. doi: 10.1109/FPT.2009.5377690.

[43] Bharath Sudev, Leandro Soares Indrusiak, and James Harbin. Network-on-Chip packet prioritisation based on instantaneous slack awareness. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 227–232, 2015. doi: 10.1109/INDIN.2015.7281739.

[44] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2):105–128, 2004. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2003.07.004. URL https://www.sciencedirect.com/science/article/pii/S1383762103001139. Special issue on networks on chip.

[45] Aline Mello, Ney Calazans, and Fernando Moraes. *QoS in Networks-on-Chip – Beyond Priority and Circuit Switching Techniques*, pages 1–22. Springer US, Boston, MA, 2009. ISBN 978-0-387-89558-1. doi: 10.1007/978-0-387-89558-1_7. URL https://doi.org/10.1007/978-0-387-89558-1_7.

[46] Zhonghai Lu and Axel Jantsch. Slot allocation using logical networks for TDM virtual-circuit configuration for network-on-chip. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 18–25, 2007. doi: 10.1109/ICCAD.2007.4397238.

[47] Florian Brandner and Martin Schoeberl. Static routing in symmetric real-time network-on-chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 61–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1409-1. doi: 10.1145/2392987.2392995. URL http://doi.acm.org/10.1145/2392987.2392995.

[48] Milos Panic, Carles Hernández, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. Modeling high-performance wormhole nocs for critical real-time embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 267–278, 2016. doi: 10.1109/RTAS.2016.7461342. URL `https://doi.org/10.1109/RTAS.2016.7461342`.

[49] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 583–594, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485972. URL `http://doi.acm.org/10.1145/2485922.2485972`.

[50] A. Psarras, J. Lee, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos. PhaseNoC: Versatile Network Traffic Isolation Through TDM-Scheduled Virtual Channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):844–857, May 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2488490.

[51] Borislav Nikolic, Sebastian Tobuschat, Leandro Soares Indrusiak, Rolf Ernst, and Alan Burns. Real-time analysis of priority-preemptive NoCs with arbitrary buffer sizes and router delays. *Real Time Syst.*, 55(1):63–105, 2019. doi: 10.1007/s11241-018-9312-0. URL `https://doi.org/10.1007/s11241-018-9312-0`.

[52] Z. Lu and A. Jantsch. Tdm virtual-circuit configuration for network-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):1021–1034, Aug 2008. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2000673.

[53] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. ISSN 1063-8210. doi: 10.1109/TVLSI.2015.2405614.

[54] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard. A metaheuristic scheduler for time division multiplexed networks-on-chip. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 309–316, June 2014. doi: 10.1109/ISORC.2014.43.

[55] N. Concer, A. Vesco, R. Scopigno, and L. P. Carloni. A dynamic and distributed tdm slot-scheduling protocol for qos-oriented networks-on-chip. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 31–38, Oct 2011. doi: 10.1109/ICCD.2011.6081372.

[56] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A Time-Predictable Memory Network-on-Chip. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-69-9. doi: 10.4230/OASIcs.WCET.2014.53. URL `http://drops.dagstuhl.de/opus/volltexte/2014/4604`.

[57] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, Feb 2017. ISSN 0018-9340. doi: 10.1109/TC.2016.2595581.

[58] Xilinx. Vivado Design Suite 2016.2, 2016. URL `https://www.xilinx.com/products/design-tools/vivado.html`.

[59] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *WCET 2010*. URL `http://www.es.mdh.se/publications/1895-`.

[60] Nangate FreePDK45 Library. The Nangate Open Cell Library, 45 nm FreePDK. URL `https://projects.si2.org/openeda.si2.org/projects/nangatelib/`.

[61] MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems, 2015. URL `http://www.mango-project.eu/`.

[62] Tomás Picornell, José Flich, Carles Hernández, and José Duato. DCFNoC: A Delayed Conflict-Free Time Division Multiplexing Network on Chip. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 95:1–95:6, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6725-7. doi: 10.1145/3316781.3317794. URL `http://doi.acm.org/10.1145/3316781.3317794`.

[63] T. Picornell, J. Flich, C. Hernández, and J. Duato. Enforcing Predictability of Many-Cores With DCFNoC. *IEEE Transactions on Computers*, 70(2):270–283, 2021. doi: 10.1109/TC.2020.2987797.

[64] J. Duato and T. M. Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1219–1235, 2001.

[65] Sebastian Kehr, Eduardo Quiñones, Dominik Langen, Bert Böddeker, and Günter Schäfer. Parcus: Energy-aware and robust parallelization of AUTOSAR legacy applications. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 343–352, 2017. doi: 10.1109/RTAS.2017.4. URL `https://doi.org/10.1109/RTAS.2017.4`.

[66] Mladen Slijepcevic, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Time-randomized wormhole nocs for critical applications. *J. Emerg. Technol. Comput. Syst.*, 15(1), January 2019. ISSN 1550-4832. doi: 10.1145/3281029. URL `https://doi.org/10.1145/3281029`.

[67] Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. ISSN 0360-0300. doi: 10.1145/3131347. URL `http://doi.acm.org/10.1145/3131347`.

[68] T. Picornell, J. Flich, D. Jose, and C. Hernández. HP-DCFNoC: High Performance Distributed Dynamic TDM Scheduler Based on DCFNoC Theory. *IEEE Access*, 8: 194836–194849, 2020. doi: 10.1109/ACCESS.2020.3033853.

[69] Douglas R. Stinson. *Cryptography : Theory and practice / Douglas R. Stinson.* The CRC Press series on discrete mathematics and its applications. CRC Press, 1995. ISBN 0849385210.

[70] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. *SIGPLAN Not.*, 53(2):751–766, March 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173191. URL `https://doi.org/10.1145/3296957.3173191`.

[71] A. Strano, D. Bertozzi, F. Triviño, J. L. Sánchez, F. J. Alfaro, and J. Flich. OSR-Lite: Fast and deadlock-free NoC reconfiguration framework. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 86–95, 2012. doi: 10.1109/SAMOS.2012.6404161.

[72] J. Flich and J. Duato. Logic-Based Distributed Routing for NoCs. *IEEE Computer Architecture Letters*, 7(1):13–16, 2008. doi: 10.1109/L-CA.2007.16.

[73] J. Flich, S. Rodrigo, and J. Duato. An efficient implementation of distributed routing algorithms for nocs. In *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, pages 87–96, 2008. doi: 10.1109/NOCS.2008.4492728.