ÉCOLE
POLYTECHNIQUE
DE BRUXELLES
UNIVERSITÉ LIBRE DE BRUXELLES

Protocols and algorithms for secure Software Defined Network on Chip (SDNoC)

Thesis presented by Soultana Ellinidou
in fulfilment of the requirements of the PhD Degree in Engineering Sciences and Technology (Doctorat en Sciences de l'Ingénieur et Technologie)
Academic year 2020-2021

Supervisor : Prof. Jean-Michel Dricot
Co-supervisor : Prof. Olivier Markowitch

Thesis jury :

Dr. Gaurav Sharma (Université libre de Bruxelles, Chair)
Prof. Jean-Michel Dricot (Université libre de Bruxelles, Secretary)
Prof. Guy Gogniat (Université Bretagne Sud)
Prof. Olivier Markowitch (Université libre de Bruxelles)
Prof. Dragomir Milojevic (Université libre de Bruxelles)
Prof. Kris Steenhaut (Vrije Universiteit Brussel)

Abstract

Author — Soultana Ellinidou
PhD Degree — Engineering Sciences and Technology
Academic Year — 2020–2021
Title — Protocols and algorithms for secure Software Defined Network-on-Chip (SDNoC)
Abstract — Under the umbrella of Internet of Things (IoT) and Internet of Everything (IoE), new applications with diverse requirements have emerged and the traditional System-on-Chips (SoCs) were unable to support them. Hence, new versatile SoC architectures were designed, like chiplets and Cloud-of-Chips (CoC). A key component of every SoC, is the on-chip interconnect technology, which is responsible for the communication between Processing Elements (PEs) of a system. Network-on-Chip (NoC) is the current widely used interconnect technology, which is a layered, scalable approach. However, the last years the high structural complexity together with the functional diversity and the challenges (QoS, high latency, security) of NoC motivated the researchers to explore alternatives of it. One NoC alternative that recently gained attention is the Software Defined Network-on-Chip (SDNoC). SDNoC originated from Software Defined Network (SDN) technology, which supports the dynamic nature of future networks and applications, while lowering operating costs through simplified hardware and software. Nevertheless, SDN technology designed for large scale networks. Thus, in order to be ported to micro-scale networks proper alterations and new hardware architectures need to be considered.

In this thesis, an exploration of how to embed the SDN technology within the micro scale networks in order to provide secure and manageable communication, improve the network performance and reduce the hardware complexity is presented. Precisely, the design and implementation of an SDNoC architecture is thoroughly described followed by the creation and evaluation of a novel SDNoC communication protocol, called Micro-LET, in order to provide secure and efficient communication within system components. Furthermore, the security aspect of SDNoC constitutes a big gap in the literature. Hence, it has been addressed by proposing a secure SDNoC Group Key Agreement (GKA) communication protocol, called SSPSoC, followed by the exploration of Byzantine faults within SDNoC and the investigation of a novel Hardware Trojan (HT) attack together with a proposed detection and defend method.

Keywords — Software Defined Network-on-Chip, NoC, routing algorithms, Hardware Trojan, Byzantine Faults, Group Key Agreement

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor, Prof. Jean-Michel Dricot and my co-advisor, Prof. Olivier Markowitch, for offering me this opportunity to work on the SOFIST research project, for their insightful advices and guidance, for reviewing this PhD thesis and also my research papers and journals during the last 3 years. Furthermore, I would like to personally thank my advisor, whose door was always open for research discussions, for his encouragement and full support during my PhD.

Secondly, I would like to thank an important person, colleague and friend, Dr. Gaurav Sharma, with whom I had the honor to collaborate and work in the same research project. I would like to thank him for supporting me despite numerous obstacles and tough situations during my PhD. Without his motivation and insights this work would have never been complete.

Thirdly, I would like to thank Prof. Guy Gogniat for giving me the opportunity to visit the Lab-STICC and interact with his team members, for his help and his valuable inputs which helped me tremendously in framing this work.

I would like to acknowledge the Master students: Theofanis Rigas, Tristan Vanspouwen and Adil Layach, who I had the opportunity to collaborate with and supervise during their Master thesis. Thank you for your contributions to my research work.

I would also like to thank my colleagues and former or current members of Optique, Photonique, Electromagnétisme, Radio communications et Acoustique (OPERA) Wireless Communication Group (WCG) : Prof. Philippe De Doncker, Prof. François Horlin, Natascha Vander Heyden, Dr. Jean-François Determe, Dr. Trung-Hien Nguyen, Dr. François Rottenberg, Dr. Utkarsh Singh, Sullivan Derenne, Alexey Garcia Padilla, Shaghayegh Monfared, Hasan Can Yildirim, Evert Ismael Pocoma Copa, Guylian Molineaux and Laurent Storrer for their nice conversions about research but

also about life during every day lunches at the lab, for their help in order to survive in Brussels and for their positivity about PhD and life in general. Furthermore, I would also like to thank my colleagues from the Quality and security of information systems (QualSEC) research group: Prof. Yves Roggeman, Dr. François Gerard, Dr. Suman Bala, Dr. Veronika Kuchta, Dr. Liran Lerman, Dr. Rajeev Anand Sahu, Dr. Gaurav Sharma and Dr. Dimitrios Sisiaridis for the nice seminars that they organised during the academic year, for sharing their research and for all research discussions that we had.

In addition, I would like to thank my beloved parents Lazaros Ellinidis and Olga Papadopoulou, my brother Nikolaos Ellinidis and my two sisters Marina and Eleni Ellinidou for their support and their long phone calls during all this time. I would also like to express my strong gratitude to my grandfather Arxelaos Papadopoulous for his financial support in the start of my PhD. Also, I could not have completed this dissertation without the support of my best friends from Greece: Antonios Ventouris, Irene-Maria Tabakis, Paraskevi Smiari, Konstantina Banti, Zoi Ztoupa, Androniki Prokopidou and also my new friends from Belgium: Shaghayegh Monfared, Elliana Lamprianidou, Stefania Traettino, Francesca Di Matteo and Riccardo Pace, who provided stimulating discussions as well as happy distractions to rest my mind outside of my research.

Furthermore, I would like to thank a very important person, Guylian Molineaux, for always being there the last months, offering unconditional help and support during these hard times of pandemic but also for reviewing my thesis and definitely for making my life better and brighter.

Finally, I would like to acknowledge that this research would not have been conducted without the financial support of Project ARC (Concerted Research Action) of Fédération Wallonie-Bruxelles.

Table of Contents

List of Figures

List of Tables

List of Acronyms

MS	Mean of Squares
$SS_{between}$	Sum of Squares between
SS_{total}	Sum of Squares total
SS_{within}	Sum of Squares within
AA	Always Active
ACC	Accuracy
ACK	ACKnowledgement
AES	Advanced Encryption Standard
AMBA	Advanced Micro-controller Bus Architecture
ANOVA	ANalysis Of VAriances
API	Application Programming Interface
AUC	Area Under the Curve
BFT	Byzantine Fault Tolerance
CA	Certification Authority
CIANAA	Confidentiality, Integrity, Authentication, Non-repudiation, Availability, Authorization
CMP	Chip Multi-Processor
CoC	Cloud-of-Chips
CPU	Central Processing Unit
CS	Circuit Switching
DB	Destination Based
DOR	Dimension Order Routing
DoS	Denial of Service
DTMA	Dynamic Task Mapping Algorithm
DyAD	Dynamically Adaptive and Deterministic
FDR	False Discovery Rate
FIFO	First In First Out

FN	False Negative
FNR	False Negative Rate
FOR	False Omission Rate
FP	False Positive
FPR	False Positive Rate
GCM	Galois Counter Mode
GKA	Group Key Agreement
GPP	General Purpose Processor
GPU	Graphical Processing Unit
HT	Hardware Trojan
HT-DoS	Harware Trojan Denial of Service
IC	Integrated Circuits
IMEC	Interuniversity Microelectronics Centre
IoE	Internet of Everything
IoT	Internet of Things
IP	Intellectual Properties
IV	Intialization Vector
MANET	Mobile Ad-hoc NETwork
MANGO	Message-passing Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces
ML	Machine Learning
MPN	Multi-Physical Network
MPSoC	Multi Processor System-on-Chip
N	Negative
NF	Negative First
NI	Network Interface
NL	North Last
NM	Network Manager
NoC	Network-on-Chip
NPV	Negative Predicted Value
OE	Odd Even
OESL	Odd Even with SeLection
ONF	Open Networking Foundation
OSI	Open Systems Interconnection
OVS	OpenVSwitches
P	Positive

PBC	Pairing Based Cryptography
pBFT	practical Byzantine Fault Tolerance
PCB	Printed Circuit Board
PCs	Personal Computers
PDF	Probability Density Function
PE	Processing Element
PK	Private Key
PKC	Public Key Cryptography
PKG	Private Key Generator
PKI	Public Key Infrastracture
PL	Physical Links
PPV	Positive Predicted Value
PSK	Pre-Shared Key
QNoC	Quality of service Network-on-Chip
QoS	Quality of Service
ROC	Receiver Operating Characteristic
SDN	Software Defined Network
SDNoC	Software Defined Network-on-Chip
SoC	System-on-Chip
SSPSoC	Secure Sdn-based Protocol over mpSoC
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privileges
TCP	Transmission Control Protocol
th	threshold
tir	traffic injection rate
TLS	Transport Layer Security
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
TSV	Through Silicon Via
TTL	Time To Leave
VC	Virtual Channels
VLSI	Very Large Scale Integration
WF	West First
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Background

Since the 90's, the number of transistors that was able to fit into a single piece of silicon increased in a predictable way, known as Moore's law [Moore, 1998]. This had as a result the digital evolution of minicomputers to Personal Computers (PCs), afterwards to smart-phones and to cloud. By placing more and more transistors into each generation of their microchip and simultaneously making them more powerful and able to support the dynamic nature of today's applications (for example in automotives [Hubner et al., 2005] and avionics [Hilbrich and van Kampenhout, 2010]).

Under the umbrella of Internet of Things (IoT) [Atzori et al., 2010] and Internet of Everything (IoE) [Miraz et al., 2015], a big variety of applications emerged in order to satisfy people's needs in transportation, healthcare, manufacturing, and energy management. All these new applications had diverse requirements, which traditional System-on-Chip (SoC) [Rajsuman, 2000] were not always capable to support due to the cost of semiconductor processing, fabrication and the complexity in terms of the amount of circuit elements for a large die [Sethi and Sarangi, 2017]. At the same time the smallest features of transistors reached 7nm [Wu et al., 2016] and Interuniversity Microelectronics Centre (IMEC) manufactured the first 3nm transistor [Cadence, 2018]. Furthermore, a huge increase in Integrated Circuits (IC) cost is observed.

Hence the chipmakers start to look for alternative ways. The current top notch approach, which the industry is investigating, are chiplets on a substrate to reduce the cost of complex semiconductor solutions, since the fabrication of large monolithic dies will become more costly. Chiplets

were introduced to break a conventional monolithic SoC into smaller pieces. More precisely, chiplets refer to the independent constituents which make up a large chip, consisting of multiple smaller dies. The need to employ multiple chips comes from reticle limit which dictates the maximum size of chips possible to be fabricated. Designs that exceeded the reticle limit had to be split up into smaller dies. The idea is that individual Central Processing Unit (CPU), memory, and other Processing Elements (PEs) will be mountable onto a relatively large slice of silicon, called an active interposer. An interposer is a thick silicon layer, which includes interconnects and routing circuits. Recently the chiplet approach has gained attention from academia [Iyer, 2016,Kannan et al., 2015], industry [Sutardja, 2015,Vijayaraghavan et al., 2017,Arunkumar et al., 2017] but also from government agencies [Seemuth et al., 2015].

A chiplet-based system is depicted in Figure 1.1. It consists of chiplets that are placed on the interposer, routing inter-chiplet connections through metal layers in the interposer, and Through Silicon Vias (TSVs) in the interposer to connect chiplet C4-bumps to the package-level interconnect. A TSV in the bottom die provides external I/O access and power delivery to the top die.

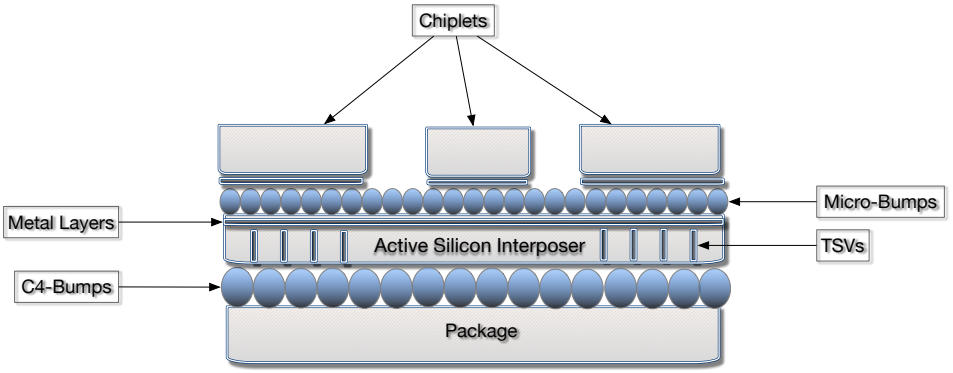


Figure 1.1: Chiplet-based System

Another novel architecture able to support the dynamic nature of the future application is referred to as Cloud-of-Chips (CoC) [Bousdras et al., 2018], in analogy to the unlimited scalability of cloud computing paradigm. The CoC consists of a large amount of interconnected IC and IC cores, which can have different communication speeds and hierarchy levels. The CoC template architecture follows a flexible architecture which can change its characteristics, such as routing logic, transmission path, priorities and

IC clustering. The template architecture and computing clusters are coupled at design time, while the communication scheme and security features are dealt with at run-time. Figure 1.2 presents a Printed Circuit Board (PCB) that hosts a package of multiple identical PEs, where each one may have many functional Intellectual Properties (IP) cores. Moreover, these PEs could be a Graphical Processing Unit (GPU), crypto processor, accelerator or a combination of such IP blocks.

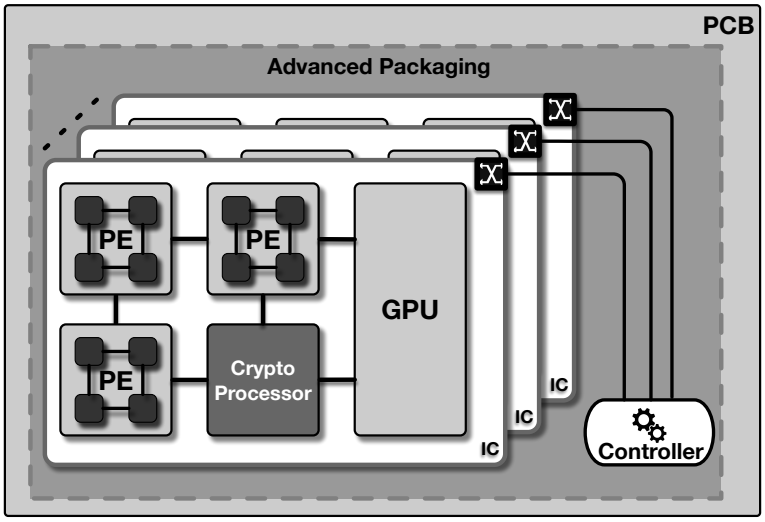


Figure 1.2: Cloud-of-Chips Platform Architecture

While, the design of chiplets and CoC have already been explored [Iyer, 2016, Vijayaraghavan et al., 2017, Bousdras et al., 2018], the interconnect fabric connecting the nodes of the entire system has been neglecting. Hence, the interconnect must be equally explored in order to enable the properly distribution of the data within the system. In case of the chiplets, each individual chiplet may contain its own local interconnect, which operates for intra-chiplet traffic and different hierarchical layers of communication should be introduced. In case of CoC scalable and highly-configurable communication infrastructures are needed.

Although current multi-chiplet architectures utilize passive integration technologies such as silicon interposers, in this research the chiplet-based SoCs that are based on active silicon interposer (Figure 1.1) have been taken into account. Despite the high interest into the passive substrates (only wires but not logic)) [NVIDIA, 2016], there is available research in academia [Kannan et al., 2015], industry and government [Beyne and

Manna, 2013] focused on active interposers. An active interposer (transistors thus logic) implements its own interconnect in order to enable the communication of the chiplets. While connecting several interconnects together, new resource cycles that cause cyclic dependencies across the chiplets can be introduced by causing deadlocks and livelocks.

A key component of the SoC is the on-chip interconnect, which is considered as the backbone, ensuring the communication of all PEs of the system. The history of the interconnect consists of three phases, which are depicted on the Figure 1.3. The first phase was driven by bus technology, with the first de-facto commercial standard being ARM's Advanced Micro-controller Bus Architecture (AMBA) [Flynn, 1997], which provides a shared communication medium for data transfer within the system. As SoCs grew in numbers of IP blocks, the bus showed its limitations (poor performance, high power consumption and big delay). In order to overcome these limitations, the crossbar interconnect was introduced in the 90's, providing high-speed point-to-point data transfer between cores [Niehaus et al., 1989]. The crossbar is suitable for a small number of nodes but not scalable as wire cost becomes even more expensive with many cores. Afterwards in the early 2000's Network-on-Chip (NoC) was introduced [Benini and De Micheli, 2002] in academia and thereafter in industry. NoC provides parallel communication but without significant cost overhead as in the crossbar interconnect. Contrary to bus and crossbar interconnects, NoC provides path diversity as several paths exist between source and destination cores by solving the scalability problem of bus-based architecture. More precisely, people referred to as an unification of on-chip communication solutions, which consists of an on-chip packet or circuit switched micro-network of interconnects. PE access the network by means of proper interfaces, and have their packets forwarded to destination through a multi-hop routing path. The scalable and modular nature of NoC together with their support for efficient on-chip communication lead to NoC-based multi-processor systems characterized by high structural complexity and functional diversity.

However, the complexity of the current NoCs and the novel hardware architecture designs motivated researchers to explore alternatives of it [Cong et al., 2014, Berestizshevsky et al., 2017]. One NoC alternative that gained attention the last years is the Software Defined Network-on-Chip (SDNoC). The Software Defined Network (SDN) technology emerged to support the dynamic nature of future network functions and intelligent applications while lowering operating costs through simplified hardware, software and management. Although SDN appeared as a research concept in 2008 [McK-

eown et al., 2008], it quickly gained significant attention from the industry over the past few years. In fact, Google, Facebook, Yahoo, Microsoft, Verizon, and Deutsche Telekom funded the Open Networking Foundation (ONF) and in this way adopted the SDN through open standards development.

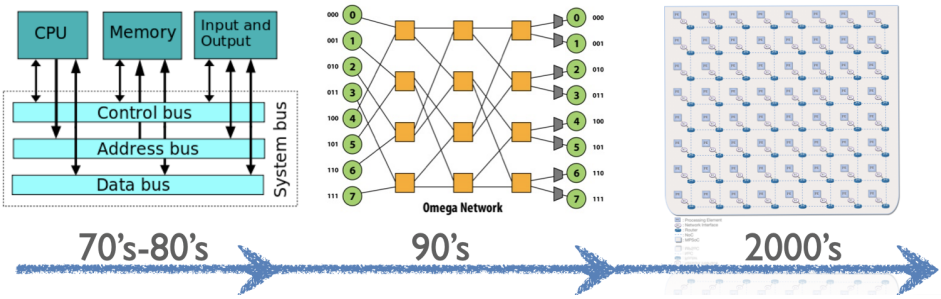


Figure 1.3: History of interconnect technology: From Bus to NoC

The approach proposed by the SDN paradigm is that data travels across multiple network entities (switches or routers) and efficient and effective data transfer is supported by a centralized controller. The controller can implement different communication rules to define the paths, as Quality of Service (QoS), fault-tolerance, and security. As far as the SoC architectures, they may adopt the SDN paradigm due to its advantages: reduced hardware complexity, high re-usability, and flexible management of communication policies. Though, the are also some challenges. SDN may be the overhead for defining the paths in software compared to hardware-based approaches and the controller can be a single point of failure of the whole system.

1.2 SDNoC integration within chiplet-based systems

The typical intra-chiplet data communication is managed by NoC, which supports regular interconnected topologies. However, in order to manage routing inside a chiplet with multiple cores, the size of routing tables will be too large enough to be accommodated on ordinary NoC routers. The memory overhead for routing tables will grow by $n^2 * k$ units where n^2 is the number of PEs on each chiplet and k is the number of chiplet on each package. Therefore, to achieve secure inter-chiplet and intra-chiplet communication on a package, some techniques based on SDN paradigm could be designed. The SDN concept came into the micro-scale networks recently,

as it is presented in the previous section, and it is still limited under research. Traditional routing mechanisms employ NoC hardware routers to manage the routes among chiplets. However, recent SDN-based strategies implement a controller with global view that controls the routing in an adaptive manner. The proposed SDNoC network for chiplet-based system is depicted in Figure 1.4. Since in this research the main focus is to cover the intra-chiplet communication and leave the inter-chiplet for future work, the controller will be placed inside a chiplet and attached to one router. The rest of the routers within the network will communicate in order to ask for a possible route for the upcoming packets. In this way the SDNoC approach is enabled. Regarding the inter-chiplet communication an extra NoC is placed on the active interposer and it is able to efficiently interconnect them.

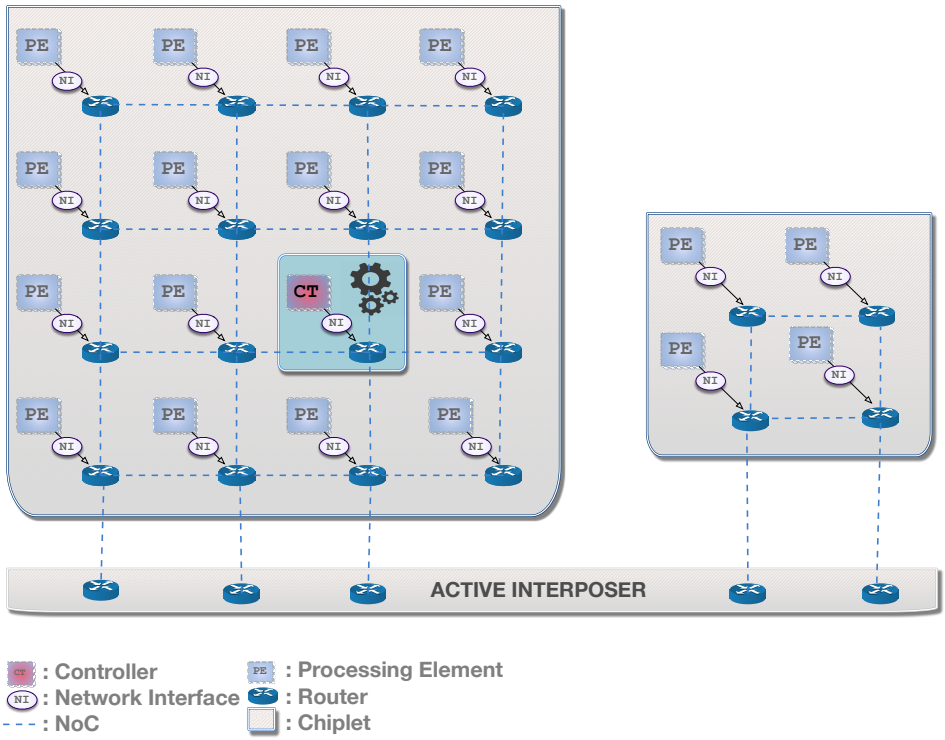


Figure 1.4: SDNoC architecture within a chiplet

1.3 SDNoC integration within CoC

To enable the reconfigurability at the different levels of CoC platform but also in order to provide quick and secure communication, the SDNoC interconnect technology is proposed. Each IC integrates a software programmable controller. All controllers are reporting to the central hardware controller. The two levels of hierarchy enable the efficient communication on the IC level as well as the PCB level. The packet forwarding is managed in an SDN way. The source IP core forwards the packet header to the on-board controller and the controller sends back the whole sequence of exit ports at each NoC router. The controllers on each IC also maintain flow tables and group tables for outside IC communication. The central controller is looking at the global view of the topology and is responsible for the updates of flow entries on these controllers. Once the flow entry is updated, the header packet is assigned a route and rest of the packets will follow the same route.

1.4 Security Challenges

As the number of processing cores is increasing dramatically, the communication among them is of high importance. NoC has direct access to all resources and information within a SoC, rendering it appealing to attackers. Precisely, the sensitive information flow on the interconnect leaves the system vulnerable to various threats. Most of the real-time applications do not support any encryption or authentication strategy to protect this information. The interface to external devices makes the IP cores more vulnerable to attacks. Moreover, frequent reconfiguration and wireless communication causes the situation to be more opportunistic. Additionally, running an untrusted application can render IP cores and routers behavior malicious. The infected IP cores extract sensitive information stored locally and forward them to some external entity. The infected router can cause arbitrary deviation from its specification, packet redirection, packet modification, (partial) packet dropping, deadlocks or livelocks.

As it previously mentioned, recent advancement leads to applicability of SDN on micro-architectural level, namely SDNoC. However, the idea is explored from the hardware and partially from networking view point, the security view point has been undermined. Hence, in this thesis the security issues of SDN by providing secure protocols in order to ensure the secure communication and by designing secure algorithms in order to detect and

defend the network from security bridges have been addressed.

1.5 Objective-Contributions

The work of this thesis is part of Self-Organising circuits For Interconnected, Secure and Template computing (SOFIST) research project supported by Project ARC (Concerted Research Action) of Fédération Wallonie-Bruxelles. This project aims the development of a future SoC architecture, which can have different communication speeds and hierarchy levels. As part of the project the design of the proposed architecture was investigated from four different perspectives: reconfigurable hardware design, real-time scheduling, flexible communication and required security primitives. In this thesis the two latter perspectives were investigated.

By taking into account the main challenges of the current interconnect technologies, which mentioned before, the main objective of this thesis is to embed the SDN concept into micro-scale systems in order to provide secure and manageable communication, improve the network performance and reduce the hardware complexity. More precisely this thesis provides 3 main contributions:

Contribution 1: Design and implementation of an SDNoC architecture [Ellinidou et al., 2018]. (Chapter 3, Chapter 4)

Contribution 2: Creation of a novel SDNoC communication protocol in order to provide secure and efficient communication between routers and controller. Precisely, MicroLET is the first SDNoC communication protocol destined for future SoC by supporting efficient routing management without significant latency [Ellinidou et al., 2019]. (Chapter 3, Chapter 4)

Contribution 3: Filling the gap of literature by exploring the security aspect of SDNoC. Firstly, the secure communication has been explored by proposing a secure SDN-based Group Key Agreement (GKA) communication protocol, followed by the exploration of Byzantine faults and the investigation of a novel HT attack together with a proposed detection and defend method [Soultana Ellinidou, 2019, Ellinidou et al., 2020b, Ellinidou et al., 2020a]. (Chapter 5, Chapter 6)

1.6 Publications

This thesis led to the following contributions in recognized international conferences and scientific journals:

Journal Articles

1. Sharma, G., Bousdras, G., **Ellinidou**, S., Markowitch, O., Dricot, J.M. and Milojevic, D., 2021. Exploring the security landscape: NoC-based MPSoC to Cloud-of-Chips. Microprocessors and Microsystems, p.103963, Elsevier.

2. **Ellinidou, S.**, Sharma, G., Rigas, T., Vanspouwen, T., Markowitch, O. and Dricot, J.M., 2019. SSPSoC: A secure SDN-based protocol over MPSoC. Security and Communication Networks, 2019, Hindawi.

3. Sharma, G., Kuchta, V., Anand Sahu, R., **Ellinidou, S.**, Bala, S., Markowitch, O. and Dricot, J.M., 2019. A twofold group key agreement protocol for NoC-based MPSoCs. Transactions on Emerging Telecommunications Technologies, 30(6), p.e3633, Wiley Online Libray.

Conference Articles

1. **Ellinidou, S**., Sharma, G., Markowitch, O., Dricot, J. M. and Gogniat, G., 2020, October. Towards NoC Protection of HT-Greyhole Attack. In International Conference on Algorithms and Architectures for Parallel Processing (pp. 309-323). Springer, Cham.

2. **Ellinidou, S.**, Sharma, G., Markowitch, O., Gogniat, G. and Dricot, J.M., 2020, October. A novel Network-on-Chip security algorithm for tolerating Byzantine faults. In 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) (pp. 1-6). IEEE.

3. **Ellinidou, S.**, Sharma, G., Kontogiannis, S., Markowitch, O., Dricot, J.M. and Gogniat, G., 2019, August. MicroLET: A new SDNoC-based communication protocol for chipLET-based systems. In 2019 22nd Euromicro Conference on Digital System Design (DSD) (pp. 61-68). IEEE.

4. Sharma, G., **Ellinidou, S.**, Vanspouwen, T., Rigas, T., Dricot, J.M. and Markowitch, O., 2019. Identity-based TLS for Cloud of Chips. In ICISSP (pp. 44-54).

5. Sharma, G., **Ellinidou, S.**, Kuchta, V., Sahu, R.A., Markowitch, O. and Dricot, J.M., 2018, August. Secure communication on noc based mpsoc. In International Conference on Security and Privacy in Communication Systems (pp. 417-428). Springer, Cham.

6. **Ellinidou, S.**, Sharma, G., Dricot, J.M. and Markowitch, O., 2018, April. A SDN solution for system-on-chip world. In 2018 Fifth International Conference on Software Defined Systems (SDS) (pp. 14-19). IEEE.

Book Chapters

1. **Ellinidou, S**., Sharma, G., Markowitch, O. Dricot, J. M. and Gogniat, G. Towards NoC Protection of HT-Greyhole Attack. In Algorithms and Architectures for Parallel Processing: 20th International Conference, ICA3PP 2020, New York City, NY, USA, October 2–4, 2020, Proceedings, Part III (p. 309). Springer Nature.

1.7 Thesis Organization

Firstly, in Chapter 2 an overview of NoC design is presented, where the architecture, topologies, routing and the challenges of NoC are explained. Afterwards, in Chapter 3, the SDN technology is described, followed by the state of the art of the SDNoC based solutions. In the same chapter, the general SDNoC is introduced together with the routing algorithms, that have been implemented, a proposed new routing algorithm, and a novel SDNoC based communication protocol. Following the previous chapter, in Chapter 4 an implementation and evaluation of SDNoC prototype is demonstrated. Precisely, in this chapter the MicroLET communication protocol is evaluated followed by a performance evaluation of different routing algorithms, which are described in the previous chapters, under different scenarios and an ANalysis Of VAriances (ANOVA) is performed in order to investigate the affect of different factors within the network perfomance. Thereafter, in Chapter 5 the security aspect of SDNoC is explored, where a novel GKA communication protocol is introduced followed by the exploration of the Byzantine faults within SDNoC and the investigation of a new Hardware Trojan (HT) attack with the proposal of a detection and defense algorithm. In Chapter 6 the implementation and evaluation of GKA communication protocol, Byzantine faults and HT attack with defense and detection algorithm are described. Finally, the conclusion and future work are presented in Chapter 7.

Chapter 2

Network-on-Chip Design

2.1 Introduction

Network-on-Chip (NoC) is a layered and scalable on-chip interconnect technology designed to replace the traditional bus and crossbar interconnects for the Multi Processor System-on-Chip (MPSoC), chiplet based systems and future System-on-Chip (SoC). NoC technology adopted concepts and techniques from large scale networks. Critical parameters such as performance, power consumption and reliability along with the fundamental differences between the on-chip networks and large scale networks have shaped the research within NoC technology. As [Benini and De Micheli, 2002] mentioned, SoC can be viewed as a micro-network of components. Thus, the electrical logic and functional properties of the interconnection scheme can be abstracted.

The diversity of communication in the network is affected by architectural issues such as system composition and clustering. Figure 2.1 illustrates how system composition can be categorized along the axes of homogenity and granularity of system cores. The figure also clarifies a basic difference between NoC and networks for more traditional parallel computers; the latter have generally been homogeneous and coarse grained, whereas NoC-based systems implement a much higher degree of variety in composition and in traffic diversity. Clustering deals with the localization of portions of the system. Such localization may be logical or physical. Logical clustering can be a valuable programming tool. It can be supported by the implementation of hardware primitives in the network. Physical clustering, based on preexisting knowledge of traffic patterns in the system, can be used to minimize global communication, thereby minimizing the total cost of communicating, power and performance wise. Generally speaking,

reconfigurability deals with the ability to allocate available resources for specific purposes. In relation to NoC-based systems, reconfigurability concerns how the NoC, a flexible communication structure, can be used to make the system reconfigurable from an application point of view. Much research work has been done on architecturally-oriented projects in relation to NoC-based systems. The main issue in architectural decisions is the balancing of flexibility, performance, and hardware costs of the system as a whole. As the underlying technology advances, the trade-off spectrum is continually shifted, and the viability of the NoC concept has opened up to a communication-centric solution space which is what current system-level research explores.

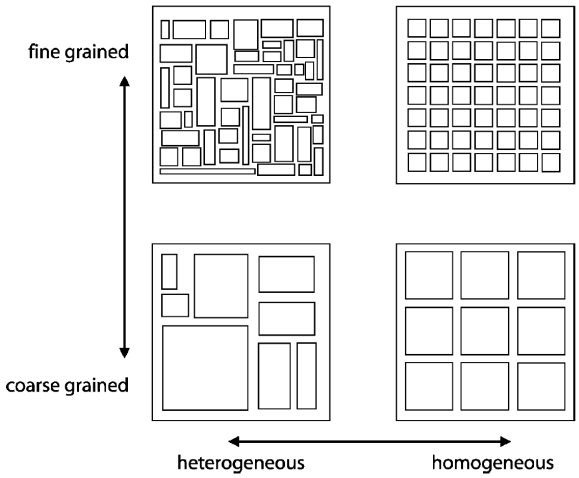


Figure 2.1: System composition categorization along the axes of homogenity and granularity of system components [Bjerregaard and Mahadevan, 2006]

In this chapter an overview of NoC design is discussed. Firstly, the NoC architecture is presented in detail, following the NoC topologies seen in literature. Afterwards, NoC routing is analyzed, which will play an important role throughout this thesis following the flow control techniques within NoC. Furthermore, an overview of the existing NoC architectures introduced in literature but also in industry is presented. Finally, the main challenges of NoC and the need for new NoC alternatives are explained.

2.2 NoC Architecture

The NoC architecture consists of 4 main core components: routers, Network Interface (NI), Processing Element (PE) and Physical Links (PL).

Routers play a key role of routing packets from source to destination nodes in the network, while the links are sets of wires that connect the routers together. A router is linked to every PE (which could be a memory, a core or a processor) and interconnects them through physical links. The NI is the intermediate entity between a PE and a router. The manner in which the routers are placed in the network is called the topology. Popular NoC topologies include: mesh, torus, ring, butterfly and binary tree [Philip et al., 2014], which are explained in the next section. However different topologies have been introduced over the years, a suitable NoC topology can be chosen for a specific application based on the performance evaluation and area consumption [Chen et al., 2011].

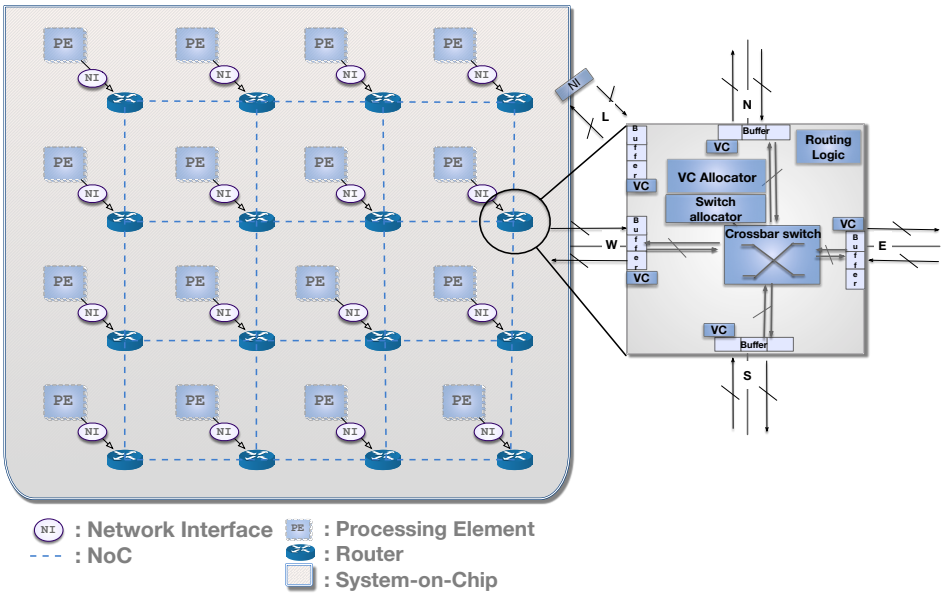


Figure 2.2: NoC architecture

In Figure 2.2 a 2D-Mesh NoC architecture is depicted, whose network entities are the following:

- **Routers**: In Figure 2.2 the architecture of a NoC 5-port router employing virtual channel flow control and switching is illustrated. The five ports correspond to the four cardinal directions (North (N), South (S), West (W), East (E)) and the local direction which connects the router with the PE through NI. The router consists of four components: the Routing Logic, the Virtual Channels (VC), the Buffers, the Switch Allocator, the VC Allocator and the Crossbar switch. It

employs a pipelined design with speculative path selection to improve performance. The router is characterized by a two-stage, pipelined architecture. The first stage is responsible for routing, where the router runs a routing algorithm in order to determine the exit port of the incoming packet. The second stage is responsible for crossbar traversal. The functionality of the router is described with respect to a 2D mesh interconnect.

- **Physical Links**: The communication between routers is managed by dedicated links, which interconnect them. Through the links, packets or flits are forwarded between routers.

- **Network Interface** is composed by two First In First Out (FIFO) memories, one logic block to interface with the network, called router adapter, and a logic block to interface with the processing unit (or core), called core adapter. The router adapter is a logic block that interacts with the network dealing with the signals of physical channel and integrates the data that come from the network, to be delivered to the core. The core adapter also is a logic block that is connected with the core, and prepares the data that comes from the core to be written in the network, concatenating the fields control bit, origin address and destination address to the data field.

2.3 NoC topologies

One of the challenges in the NoC design is the choice of the best topology to meet the bandwidth and latency requirements for the target application with the lowest power and area cost. Different topologies are proposed in the literature for the design of NoC [Philip et al., 2014], some of them are the following:

- **Mesh**: A mesh topology consists of m columns and n rows of nodes. In a mesh, nodes are connected as a grid, as shown in Figure 2.3a. Addresses of routers and resources can be easily defined with the coordinates (x, y).

- **Torus**: A torus architecture (Figure 2.3b) is obtained by adding direct connections to two end nodes in the same row or column in a mesh architecture. Compared with mesh, its diameter is reduced. A regular torus has long wrap-around links. However, by folding a torus, long wires can be avoided.

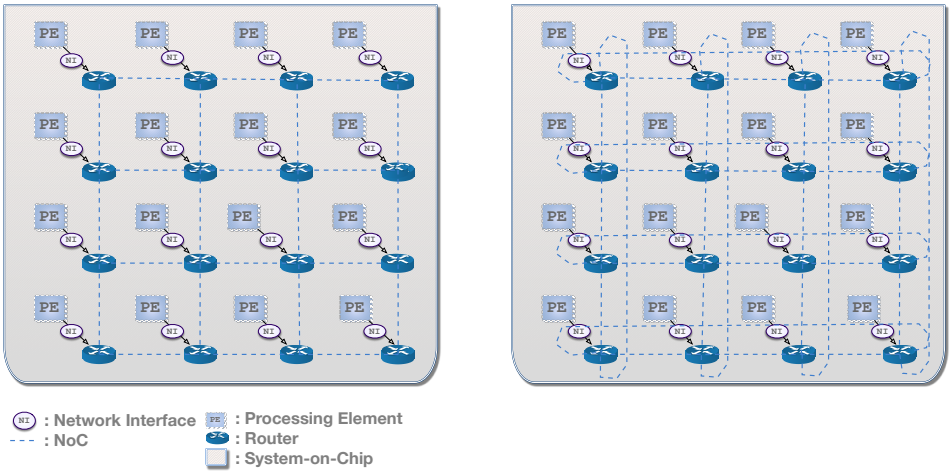


Figure 2.3: NoC architecture: a) Mesh, b) Torus.

- **Ring**: In a ring architecture, all nodes are connected in a ring, as shown in Figure 2.4a. Every node has two neighbors regardless the size of the ring. The small degree is preferable, but the diameter increases linearly with the number of nodes.

- **Star**: A star network (Figure 2.4b) consists of a central router in the middle of the star, and computational resources or subnetworks in the spikes of the star. The capacity requirements of the central router are quite large, because all traffic between the spikes pass through it. This causes a large possibility of congestion in the middle of the star.

- **Tree**: In a tree topology nodes are routers and leaves are computational resources. The routers above a leaf are called as leaf's ancestors and correspondingly the leafs below the ancestor are its children. In a fat tree topology, each node has replicated ancestors which means that there are many alternative routes between nodes (Figure 2.5a).

- **Butterfly**: A butterfly network (Figure 2.5b) is unidirectional or bidirectional. For example a simple unidirectional butterfly network contains 8 input ports, 8 output ports and 3 router levels which each contains 4 routers. Packets arriving to the inputs on the left side of the network are routed to the correct output on the right side of the network [Arjomand and Sarbazi-Azad, 2008]. In a bidirectional butterfly network, all the inputs and outputs are on the same side of the network. Packets coming to inputs are first routed to the other side of the network, then turned around and routed back to the correct output. Butterfly networks have low latency and have higher bandwidth

than other topologies however it lacks of path diversity and they have higher number of links.

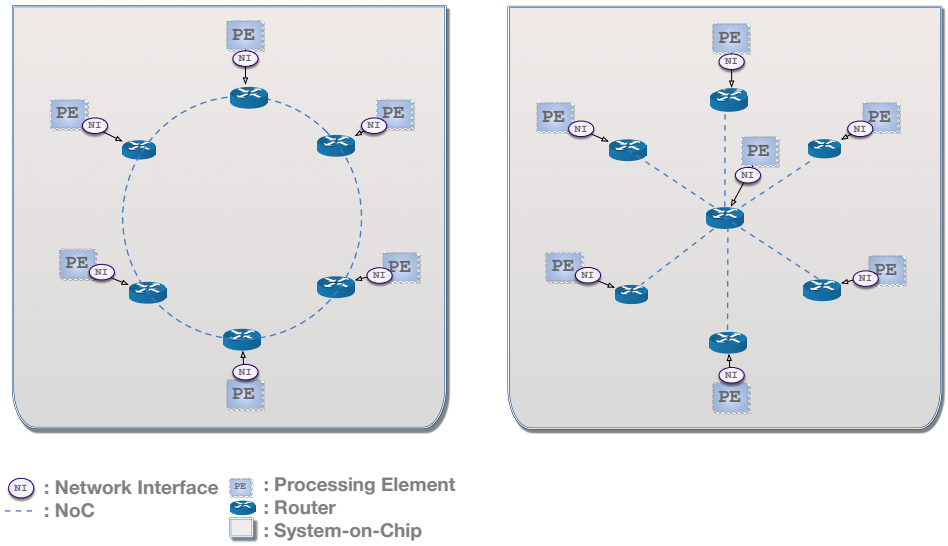


Figure 2.4: NoC architecture: a) Ring b) Star.

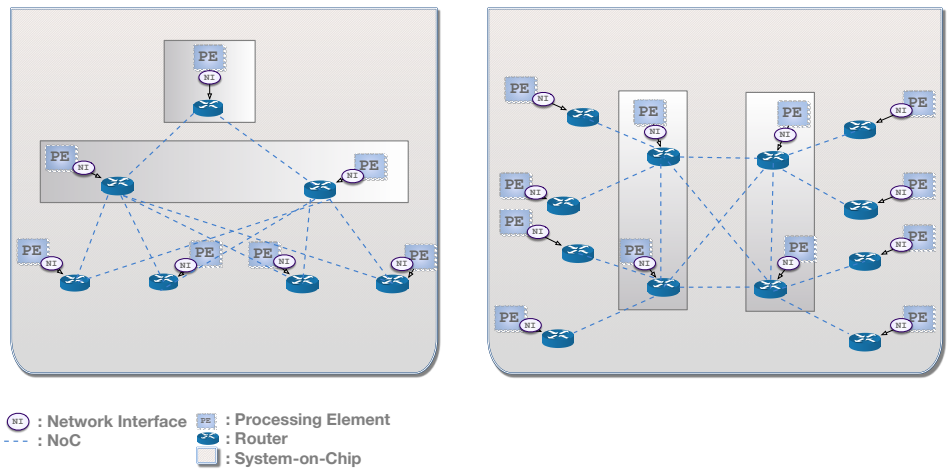


Figure 2.5: NoC architecture: a) Tree b) Butterfly.

2.4 NoC Routing

Routing is a process of selecting a path from a source to a destination node within a network or between different networks. Routing within NoC is similar to routing in any network. Specifically, routing algorithms are divided into two main groups (Figure 2.6): the oblivious and the adaptive algorithms. Oblivious algorithms are divided into two subgroups: deterministic and stochastic algorithms. Adaptive algorithms are also divided into two subgroups: minimal and fully adaptive algorithms [Rantala et al., 2006].

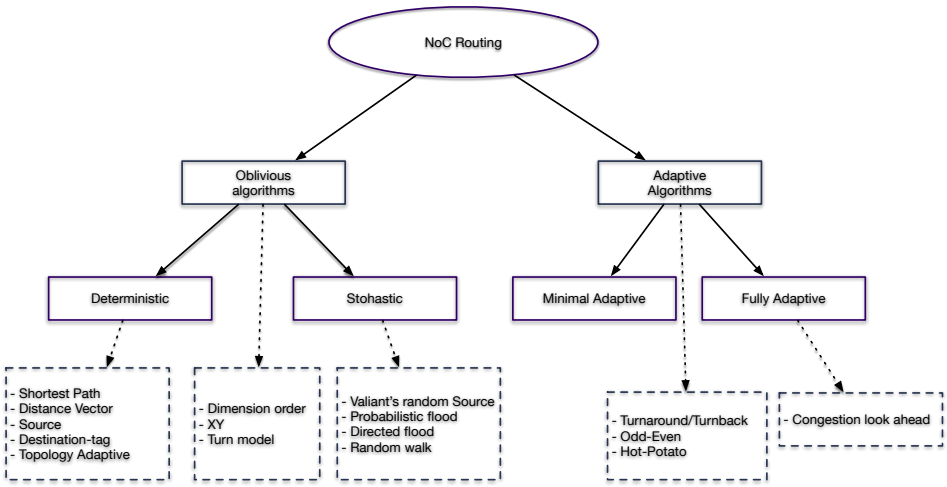


Figure 2.6: Routing levels in NoC

In oblivious algorithms, the route of the packets is determined by the source and the destination node. In deterministic algorithms, the same path is always chosen and in the stochastic algorithms a random route is chosen between a source and a destination node.

In the case of *deterministic algorithms*, a *shortest path routing* is the simplest algorithm, where packets are always routed along the shortest possible path. A distance vector routing and a link state routing are shortest path routing algorithms. In *distance vector* routing, each router has a table, which contains updated information about neighbor routers. This table is exchanged with other routers, which take routing decision by counting the shortest path on the grounds of their routing tables. The *link state routing* is a modification of distance vector routing. The basic idea is the same as in distance vector routing, however each router shares its rout-

ing table with every other router in the network. In the case of link state routing in a NoC, routing tables covering the whole network are stored in a router's memory during the production stage and the routers use their routing table updating mechanisms in the case of changes in the network structure or in the case of faults [Ali et al., 2005]. In *source routing*, the source router makes decision about a routing path of a packet by storing the path in the header of a packet before sending it to the routers along the route. In case of *destination-tag routing*, the source router stores the address of the destination, known as a destination-tag, in the header of a packet. However, every router makes routing decisions independently of the address of the receiver. Deterministic routing algorithms can be improved by adding some adaptive features to them. Following this concept a *topology adaptive routing* algorithm is introduced where a systems administrator can update the routing tables of the routers in certain circumstances.

In the case of *stochastic algorithms* the most common type are the flooding algorithms. The simplest algorithm is *probabilistic flooding*, in which routers send a copy of an incoming packet to all possible directions (flooding) and at least one of the packets will arrive at its destination. A *directed flood routing* algorithm is an improved version of the probabilistic one, where packets are directed in the direction of the destination. Another flooding algorithm is the *random walk* algorithm, where a predetermined amount of packet copies are forwarded to the network and every router along the routing path sends the incoming packets through some of its output ports. Another stochastic algorithm is the *Valiant's random* algorithm, which equalizes traffic load on the network. Firstly a random intermediate node is picked to which packets are routed and afterwards the packets are routed to their destination [Dally and Towles, 2004].

An example of independent oblivious routing algorithm is the *Dimension Order Routing (DOR)*, which is a typical minimal turn algorithm. DOR determines in which direction the packets will be routed during the transmission of a packet from source to a destination [Dally and Towles, 2004]. The most used routing algorithm is the *XY*, which routes packets first along the x axis or horizontal direction to the correct column and then along the y axis or vertical direction to the destination. XY routing is well suited for a network using mesh or torus topology. Furthermore, *turn model* algorithms determine a turn or turns which are not allowed while routing packets through a network. There are multiple turn model algorithm, which will be further discussed in Chapter 3.

Due to its distributed nature, in which each node can make routing decisions independent from others, oblivious routing is widely adopted in on-chip interconnection networks. However, today's oblivious routing algorithms face difficulties with certain traffic patterns, especially when bandwidth demands of flows vary with time. This because routes are not adjusted for different applications and the route decisions are taken during the design and not run time.

In *adaptive routing*, given a source and a destination address, the route of a packet is dynamically adjusted depending on, for instance network congestion or traffic pattern. In minimal adaptive routing algorithms the routes of the packets are determined along the shortest path and in fully adaptive routing algorithms the route is chosen is always the less congested. Due to its dynamic load balancing, adaptive routing can achieve higher throughput and lower latency compared to oblivious routing.

In case of *fully adaptive* algorithms a *congestion look ahead* algorithm gets information from other routers and based on them the routing algorithm can direct packets to bypass the congestions [Kim et al., 2005].

Other adaptive algorithms include *turnaround routing* algorithms, which are mainly designed for butterfly and fat-tree network topologies. In case of source and destination routers are placed on the same side of the network, packets are first routed from source to a random intermediate node on the other side of the network. When packets arrived at the intermediate node, they are turned around and then routed to the destination on the same side of the network. The routing from the intermediate node to the destination is performed by the destination-tag routing, as previously explained. An *odd-even routing* is an adaptive algorithm used in Dynamically Adaptive and Deterministic (DyAD) NoC system. The odd-even routing is a deadlock free turn model which prohibits turns from east to north and from east to south at nodes located in even columns and turns from north to west and south to west at nodes located in odd columns [Chiu, 2000]. More details about it can be found on the Chapter 3. A *hot-potato routing* algorithm routes packets without storing them in routers' buffer memory. Hence, packets are constantly moving before they reach their destination. However if two packets are simultaneously destined to the same direction, the router directs one of the packets to another direction and there is a big possibility that the packet will be delayed before reaching to its destination [Nilsson, 2002].

2.4.1 Routing Problems

The continuous effort to increase the reliability of the network while ensuring a sensible performance constitutes the main challenges among NoC routing problems. The routing issues are diverse by having a negative impact on network performance. Specifically deadlock, livelock and starvation are potential problems on both oblivious and adaptive routing [Rantala et al., 2006]. In the next subsections the most important problems are detailed.

2.4.1.1 Deadlock

Deadlocks are considered as one of the most difficult problems in NoC routing. They occur when two (or more) packets are waiting to be routed, while they reserve the network resources (buffers, channels), both hold resources while requesting others. Typically the routers do not release the resources before they get the new ones so the routing process can not be performed and the packets are stuck causing a enormous damage to the network. Two approaches cope with deadlocks: deadlock avoidance and deadlock recovery. *Deadlock avoidance* schemes restrict the routing algorithm with some specific constraints in order to avoid deadlocks. On the other hand, *deadlock recovery* schemes try to detect and resolve the deadlock situations when occurring. Deadlock detection and recovery mechanisms are complicated to implement due to the unpredictability of deadlock situations. This is why deadlock avoidance schemes are the most widely spread.

2.4.1.2 Livelock

Livelocks occur when packets continuously flow across the network without ever reaching their destination. This is a typical phenomenon in NoCs' non-minimal adaptive routing algorithms. The classic way to avoid livelocks in the case of a distributed and non-minimal adaptive routing algorithm is to add to each packet a Time To Leave (TTL) counter which is decremented by each router it encounters and the oldest packets get priority for the output channels. On the other hand, with minimal routing, each output channel leads the packet closer to its destination. If these output channels are occupied, the packet has no choice but to wait for their release. Therefore, minimal routing is necessarily livelock-free [Dally and Towles, 2004]. It should be noted that livelocks do not occur in SDNoCs since the routes of the packets can be computed before the packets start to flow.

2.4.1.3 Starvation

Using different priorities can create a situation where some packets with lower priorities never reach their destinations. This occurs when the packets with higher priorities reserve all resources all the time. *Starvation* can be avoided by using a fair routing algorithm or reserving some bandwidth only for low-priority packets [Benini and De Micheli, 2002].

2.5 Flow Control

Flow control determines the manner in which network resources are allocated. Specifically, flow control dictates the buffer and link allocation schemes. In packet-switched NoCs, a data message is broken into a predefined packet format. A network packet can be further broken into multiple flits, which size of flit normally equals physical channel width. Additional information is added to each flit to indicate header, body, and tail flit. The routing and other control information can either be added only to the header flit or it can be added to each flit depending on implementation. However in a circuit-switched NoC, a connection (i.e. circuit) is first established between a source and destination nodes before actual data transfer takes place. Circuit switching is used for providing guaranteed services where system predictability is required.

In *store-and-forward* flow control [Dally and Towles, 2004], before forwarding the packet to the next node, the router waits until the whole packet has been transmitted into its local buffer. This means that the input buffer must have enough space to store the whole packet, which can increase router area and power consumption. This scheme also increases the communication latency, as packets spend a long time at each node just waiting for buffering, although the output port might be free.

Virtual cut-through [Kermani and Kleinrock, 1979] improves on store-and-forward flow control by allowing a packet to be routed to the next router even before the whole packet arrives at the current router. However, the packet is only forwarded if the downstream router has enough buffer space to store the complete packet. This means that buffer size remains the same as in the case of store-and-forward flow control with improvement in per-hop latency.

Wormhole routing [Seiculescu et al., 2010] is a more robust scheme, as it allocates buffer space at the granularity of flit, opposed to the virtual

cut-through and store-and-forward scheme which allocates buffers at the granularity of packet. As soon as flit of a packet arrives at an input port, it can be forwarded even if only one flit space is available in the input port of the next router (and output channel is not allocated). The wormhole flow control scheme results in low-area routers, and it is therefore widely used in most on-chip networks. The term wormhole implies that a single packet can span multiple routers at the same time. The main downside of this scheme is that the multiple links can be blocked at the same time in case the header flit of a multiple flit packet is blocked in one of the routers on the communication path.

2.6 Overview of Academic and Commercial NoCs

Æthereal [Goossens et al., 2005] is developed in Philips Research Laboratories by aiming at achieving composability and predictability in system design and eliminating uncertainties in interconnects, by providing guaranteed throughput and latency services. The Æthereal NoC consists of routers and network interfaces. The routers, use input queuing, wormhole routing, link-level flow control and source routing. The network interfaces have a modular design, composed of kernel and shells. The NI kernel provides the basic functionality, including arbitration between connections, ordering, end-to-end flow control, packetization, and a link protocol with the router. The network connections are configurable at runtime via a memory-mapped configuration port. Consequently, through Æthereal it is provided efficient network offering high-level services (including guarantees), which allows runtime network programming using the network itself. Æthereal designed for SoCs in the consumer electronics domain, particular digital TV (DTV) and set-top boxes (STB).

Tilera iMesh on-chip interconnect network [Wentzlaff et al., 2007] is an example of NoC designed for homogeneous multi-core chips. The iMesh interconnect architecture has been used in the commercial TilePro64 multi-core chip. The 72-core Tilera GX chip also uses the same NoC design. The proposed NoC architecture is different from other academic and commercially available on-chip architectures in terms of the number of physical NoC. iMesh provides five different physical NoC channels: two of these networks are used for memory access and management tasks, while the rest are user accessible. The motivation is that future integrated circuits will have enough silicon resources to integrate more than one NoC per chip.

Xpipes [Bertozzi and Benini, 2004,Dall'Osso et al., 2012] is an advanced

NoC architecture, which targets high performance and reliable communication for on-chip multi-processors. It consists of a library of soft macros (switches, NI and links) that are design-time composable and tunable so that domain- specific heterogeneous architectures can be instantiated and synthesized. Xpipes use a static routing protocol called "street sign" routing along with wormhole switching for on-chip communication and it is implemented in SystemC. Links can be pipelined with a flexible number of stages to decouple link throughput from its length and to get arbitrary topologies. Moreover, the authors proposed a tool called Xpipes Compiler, which automatically instantiates a customized NoC from the library of soft network components, in order to test the Xpipes-based synthesis flow for domain-specific communication architectures.

Message-passing Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces (MANGO) [Bjerregaard and Sparso, 2005a, Bjerregaard and Sparsø, 2006] is a clockless NoC, which targets coarse-grained type SoCs. MANGO provides connectionless routing as well as connection-oriented guaranteed services. As far as the design of MANGO, the routers implement VC as separate physical buffers. The guaranteed services connections are established by allocating a sequence of VCs through the network. While the routers are implemented using area efficient bundled-data circuits, the links implement delay insensitive signal encoding. This makes global timing robust, because no timing assumptions are necessary between routers. A scheduling scheme [Bjerregaard and Sparso, 2005b], schedules access to the links, allowing latency guarantees to be made, which are not inversely dependent on the bandwidth guarantees.

Nostrum [Millberg et al., 2004a,Millberg et al., 2004b] focused on architecture and platform-based design, targeted towards multiple application domains. The authors highlighted advantages of a grid-based, router-driven communication media for on-chip communication as a solution to high complexity of working with high density Very Large Scale Integration (VLSI) technologies. Nostrum guaranteed services, which implemented by virtual circuits, using an explicit time division multiplexing mechanism.

Quality of service Network-on-Chip (QNoC) [Bolotin et al., 2004,Dobkin et al., 2009] aims at providing different levels of QoS for the end users. The architecture of QNoC is based on a regular mesh topology and it uses wormhole packet routing. Packets are forwarded using the static XY coordinate-based routing. It does not provide any support for error correction logic and all links and data transfers are assumed to be reliable. QNoC identified

different service levels based on the on-chip communication requirements. Precisely, SoC modules are placed so as to minimize spatial traffic density, unnecessary mesh links and switching nodes are removed, and bandwidth is allocated to the remaining links and switches according to their relative load so that link utilization is balanced.

Intel used a mesh-based NoC for an 80-core TeraFlop experimental chip [Hoskote et al., 2007]. The mesh NoC uses five stage pipelined routers designed for 5 GHz frequency. This results in 1 ns per-hop latency. According to experiments conducted on the research chip, the NoC consumed about 28% of total chip power although it consumed 17% of total chip area. Furthermore Intel has also introduced a 48-core mesh NoC-based multi-core chip called single-chip cloud computer (SCC) [Howard et al., 2010]. The target frequency for the NoC was set at 2 GHz. The router is four-stage pipelined and uses virtual cut-through flow control. To mitigate the problem with higher NoC power from the previous 80-core chip, Intel opted for a different scheme for NoC. The NoC was organized as a 6×6 mesh so that two compute cores share a single router. These techniques helped to reduce the share of NoC power to 10%.

2.7 NoC challenges

As stated in Chapter 1, the NoC complexity challenges, like Quality of Service (QoS), security, latency, traffic variability and network topology, and the new hardware architectures motivated the researcher to start exploring NoC alternatives. In this section these challenges are explored.

2.7.1 Quality of Service

QoS originates from telecommunication networks, where it refers to provide system predictability or service guarantee. For NoCs, communication QoS focuses on the allocation of communication resources (routers and wires), according to the application communication characteristics, like latency and throughput. Most techniques that provide QoS are expensive in terms of design complexities [Owens et al., 2007]. A classic technique to provide QoS is by creating a connection between source and destination (i.e. circuit switching) before the actual data transfer. This method is used in some SoC architectures [Liang et al., 2000, Rijpkema et al., 2003] . However, it has been observed that this method leads to poor scalability since the router area growth is proportional to the number of required connections.

Furthermore, managing circuits introduces additional latency. Hence, new methods should be introduced in order to ensure the QoS of a NoC

2.7.2 Latency

NoC is required to provide low latency under stringent power, area and delay constraints. Therefore, minimizing delay is a crucial aspect of NoC design. A NoC router has a 4 to 5 stage pipeline which increases the latency. Techniques such as speculation have been proposed to reduce the pipeline to 1 or 2 stages. However, more study is still required to improve the accuracy and efficiency of such design techniques [Owens et al., 2007]. Adaptive routing techniques provide faster data routing in NoC, hence minimizing the delay. However, this often requires additional complexity in terms of area and resolving issues such as deadlocks, livelocks and starvation as explained in Section 2.4. Thus, techniques that provide low latency without compromising NoC area and power are an important goal in NoC design.

2.7.3 Security

NoC is the heart of data communication between processing cores in a SoC. Since it has direct access to all resources and information within a SoC, attackers have strong motivation to exploit its possible vulnerabilities. For example, packets transferred via NoC are exposed to snooping. Additionally, a Hardware Trojan (HT) (Section 5.4) can be deployed among the NoC nodes in order to apply security threats of extracting sensitive information or degrading the system performance [Daoud, 2018]. A key challenge is to provide secure and reliable communication in the SoC, even in case an untrusted NoC IP is inserted into it.

Precisely, the sensitive information flow on the NoC leaves the system vulnerable to various threats. Most of the real-time applications do not support any encryption or authentication strategy to protect this information. Hence the applications running on an SoC based platform can be equally prone to attacks. Moreover, frequent reconfiguration and wireless communication causes the situation to be more opportunistic. Additionally, running an untrusted application can turn the IP core behavior malicious. In that case the infected IP core extracts sensitive information that is transfered to it through the NoC, stores them locally and forwards them to an external entity [Sharma et al., 2018]. The basic requirements of any secure communication are confidentiality, integrity, authentication and availability. However, access control is an additional primitive to care about, in this scenario. In this research, it is not considered any of the side channel

attacks as well as physical attacks.

The threat model for NoC covers mainly three attacks as following:

- **Denial of service attack**: In order to make the NoC resources unavailable to legitimate IP cores, an attacker may launch several attacks. The possible approaches to waste resources are replay, incorrect path, deadlock and livelock.
- **Extraction of secret information**: In this case, a malicious router along the path from a given source to a given destination attempts to read some secure information. This information might be extremely critical, such as cryptographic keys used for encryption.
- **Hijacking**: In this attack, an attacker tries to write some data in a secure memory area in order to change the system behavior. The attack can be launched by using buffer overflow or reconfiguring the internal registers.

In recent literature, additional threats have been introduced. When a NoC is supplied to a SoC integrator, there is a chance of it being equipped with a HT [Rajesh et al., 2018]. In order to activate the HT, a malicious circuit is inserted during the design of the IP block or a malicious program can activate the Trojan later at runtime. The possible attacks due to infected router within NoC and their solutions are:

- **Snooping of sensitive data**: In this case the information flow between any two routers must be confidential and accessible to them only.
- **Corrupt the data**: During the routing of information, no malicious router is allowed to modify the messages. The integrity constraints must provide end to end security.
- **Spoofing**: The destination router can verify the identity of source router.
- **Denial of Service**: In case of Denial of Service (DoS) attack, algorithms should be introduced in order to find alternatives paths and make the recourses available again.

However, there are many techniques in the literature to approach the security within a SoC depending on the architecture or application or interconnect. Some of them, with respect to NoC can be listed as follows:

- Creation of security zones and protecting them via firewall around them [Fernandes et al., 2015, Grammatikakis et al., 2014, Sepulveda et al., 2014].
- Secure routing [Fernandes et al., 2016, Sepulveda et al., 2017a, Sepulveda et al., 2016]
- Secure memory access to IP cores [Fiorin et al., 2008]
- Secure communication with key agreement approach [Sharma et al., 2019, Sepulveda et al., 2017b]

2.8 Summary-Discussion

NoC interconnect is a scalable and modular technology, enabling the efficient programming of the interconnect. Its advances have made it the preferred choice for the communication backbone within SoCs. Conceptually, NoC is similar to general-purpose networks by employing a micro network stack, that encompasses different levels of abstraction. The Physical layer is responsible for the physical aspects of communication, such as wiring and the embedded logic responsible for signal processing. Architecture and Control layers employ the concept of data links and routing algorithms that establish point-to-point or end-to-end connections among the communicating elements, and encapsulate data into packets for exchange among different NoC elements. In the Software layer, system services and applications execute on top of the lower level interfaces. This paradigm decouples abstraction layers, increasing modularity and subsequent reuse of previously designed IP modules.

NoC consists of routers interconnected by links. Routers are responsible for routing packets from source to destination nodes in the network, while the links are sets of wires that connect the routers together. According to the placement of routers within the network, the NoC follows a specific topology. As previously mentioned, popular topologies include mesh, torus, and ring.

Furthermore, contrary to bus and crossbar interconnects, NoC provides path diversity as several paths exist between source and destination cores, which can be managed by the routing algorithms applied on NoC. The path diversity can be exploited to mitigate performance loss caused by high network contentions since alternative network paths can be utilized, however this requires new routing techniques to deployed within NoC.

As NoC becomes the de-facto on chip communication, a lot NoC architectures have been observed both in research but also in industry. Here, some of them were presented without including a full list of NoC implementations. Precisely, $\times$*PIPES* targets a platform-based design methodology, in which a heterogeneous network can be instantiated for a particular application. *ÆTHEREAL*, Nostrum and MANGO implement more complex features such as guaranteed services, and target a methodology which draws closer to backbone-based design. Tilera iMesh targets homogeneous multicore chips and it is already a commercial solution.

Although NoC has become the pervasive on-chip interconnect for SoCs, numerous challenges exist. NoC is a highly complex and functional diverse interconnect technology with many challenges, like decrease of QoS, high latency and many security issues. Hence, further exploration from a research aspect is needed in order to cover most of the challenges and provide less complexity within its design. One NoC alternative that gained attention the last year is the Software Defined Network-on-Chip (SDNoC). SDNoC enables extremely flexible communication infrastructure of future system and combines design-time reconfigurability of on-chip systems and highly configurable communication of macroscopic systems. More details about SDNoC can be found in the next chapter.

Chapter 3

Software Defined Network-on-Chip

3.1 Introduction

Software Defined Network-on-Chip (SDNoC) is a Network-on-Chip (NoC) communication paradigm rather than a specific design and implementation. SDNoC originates from Software Defined Network (SDN) technology, in order to support future network functions and Internet of Things (IoT) applications while lowering operating costs by simplifying the hardware, software and management. However, SDN technology was designed for large scale networks and in order to be ported into the microscale networks some proper alterations need to be considered together with a new architecture and design. The SDNoC interconnect technology attracted many researcher during the past few years.

Recently, researchers have explored the pros and cons of using the SDN paradigm for the communication of Processing Elements (PEs) within Multi Processor System-on-Chips MPSoCs. The main benefits of SDNoC are the higher flexibility during run-time, the self-adaptive network management and the reduced hardware complexity. Hence, by using the SDNoC the routers within the interconnect will no longer be overloaded with specific designs to support different features, like Quality of Service (QoS), fault tolerance and power management, instead they become configurable and at the same time they are capable to redirect NoC packets according to the SDNoC controller rules. Due to the controller's global knowledge of resources, it may adopt policies to mitigate faults, balance the communication load, secure the communication and protect the information, manage power consumption, and provide QoS.

In this chapter, firstly, the SDN concept is explained, followed by a description of the state of the art of the SDNoC based solutions in the field of SoCs, along with the contributions in the field. Afterwards a proposed general SDNoC architecture is presented. Also, different existing routing algorithms within SDNoC are described, together with a proposed new routing algorithm. Lastly, a novel SDNoC-based communication protocol is introduced.

3.2 Software Defined Network

The SDN architecture consists of three main planes as shown in Figure 3.1: Application, Control, and Data. The Data plane consists of forwarding network equipment i.e., switches.[1] The Control plane contains the controllers, that facilitate setting up and tearing down data paths in the network (Data plane) according to the requirements of the running applications (Application plane). The Control plane is linked with the data plane through an Application Programming Interface (API), referred to as the south-bound API. If multiple controllers exist, connections among them are called east and west-bound APIs. The controller-application interface is referred to as north-bound API.

The goal of SDN is to provide the ability to users to control and manage the forwarding plane (hardware) in a network through controllers. In other words, SDN exploits the ability to split the Data plane (forwarding of the packets) from the Control plane (route planning and optimization) [Hu et al., 2014]. This paradigm provides a view of the entire network, and enables global changes without a device-centric configuration on each router separately. Furthermore, the Control plane could consist of one or more controllers, depending on the size of the network. The controller can form a peer-to-peer, high-speed, reliable and distributed network control. The switches in the Data plane, forward packets among them by checking the flow tables that are controlled by the controller in the Control plane.

Regarding the communication between switches and controller, specifically in the south-bound API, there are several communication protocols that appeared recently in literature, one of the most widely used being OpenFlow [McKeown et al., 2008]. In the OpenFlow specification [Foun-

[1] For the sake of clarity, it is important to note that , unlike in traditional networking, the words "switch" and "routers" are referring to the same concept in the field of NoC,.i.e, a packet forwarding entity that interconnects processing nodes and transmit the packets along a pre-defined data path.

dation, 2015] it is mentioned that the Data plane is controlled by providing rules (flows) to the network devices (switches). Each flow entry is an instruction for matching the incoming packets with their destinations. OpenFlow establishes a unicast communication channel between each individual router and the controller. It allows the controller to discover routers, create rules for the switching hardware and also collects statistics. Since OpenFlow is layer 4 (according to Open Systems Interconnection (OSI) model) protocol, designed for large scale networks and therefore it is not adaptable in micro-scale networks due to the vast number of network messages and rules that it contains. Hence a new lightweight communication protocol should be designed in order to fulfill the needs of micro-scale networks. Moreover, the OpenFlow protocol does not enforce security as compulsory which leaves the network vulnerable to several attack scenarios [Zhang et al., 2018a].

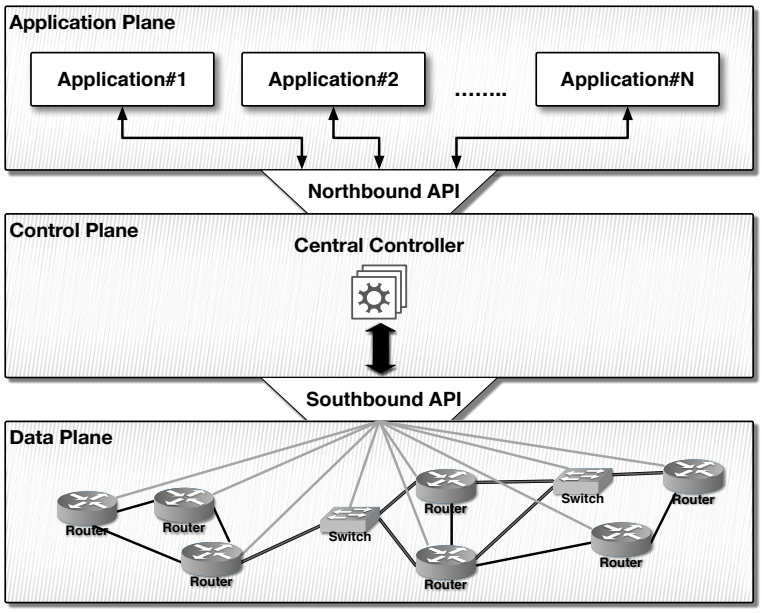


Figure 3.1: SDN Architecture

3.2.1 Security Issues

A number of SDN security analyses have recently been performed [Klöti et al., 2013, Zhang et al., 2018a], which have found that the altered entities or the links between entities in the SDN framework introduce new vulnerabilities, which were not present before. Following the data flow and interaction among SDN entities, Microsoft presents the Spoofing, Tamper-

ing, Repudiation, Information disclosure, Denial of service and Elevation of privileges (STRIDE) threat model [Hernan et al., 2006] to meet respectively the security requirements Confidentiality, Integrity, Authentication, Non-repudiation, Availability, Authorization (CIANAA). The STRIDE model attacks are listed as follows:

- **Spoofing (Authentication)**: an attacker masquerades as a legitimate user, by sending packets in order to gain access to the network.
- **Tampering (Integrity)**: an attacker attempts to deliberately modify given data from unauthorized transmissions. This could happen when the controller installs flow rules, aiming to modify or falsify data packets or flow counters [Hernan et al., 2006].
- **Repudiation (Non-Repudiation)**: an attacker claims that he did not do something or were not involved or making it impossible to link an action back to him, which violates non-repudiation. Most times, attackers do not want their identity to be known, so they hide their malicious activities to avoid being caught or blocked. Specifically, repudiation attacks occur when a system does not adopt controls to properly track user's actions, thus permitting malicious manipulation.
- **Information disclosure (Confidentiality)**: an attacker has information in his possession that are not permitted to have. In the context of SDN, this could lead to side channel attacks intended to reveal extended information about the system.
- **Denial of service (Availability)**: an attacker attempts to prevent legitimate users from accessing the service. The Denial of Service (DoS) attacks are introduced in order to make the system unavailable to receive and transmit data. In the SDN concept, the controller should be aware of the network state on a regular basis in order to apply rules, which make an SDN base system vulnerable for DoS [Yan and Yu, 2015].
- **Elevation of privilege (Authorization)**: an attacker alters his privilege to have access to the system by performing system operations. In order to perform this attack, an attacker should have access to the controller, which is considered as less likely to happen, due to the proposed use of Transport Layer Security (TLS) [Sezer et al., 2013].

The OpenFlow standard describes the use of the TLS protocol. However its use is not well enforced [Foundation, 2015]. It is written in the specification that the switch initiates a standard TLS or Transmission Control

Protocol (TCP) connection to the controller which means that the use of TLS is completely optional. In fact, security mechanisms such as TLS, protect against many attacks, however the threats should not be overlooked when moving to SDN and OpenFlow.

3.3 State of the art

As it is previously stated, SDNoC is not an alternative design of the classic NoC, but it is a new communication paradigm that tries to incorporate the SDN technology into SoCs in order to minimize the hardware complexity and provide quick and safe communication within PEs. The SDNoC concept attracted many researcher and their contributions in this specific research field start to rise through the time.

3.3.1 Literature

SDNoC concept introduced for first time in 2014 by [Cong et al., 2014], the authors proposed a novel SDN architecture, where it decouples hardware from software defined control logic, and applications are able to configure the network according to their requirements and to improve the system performance. In their architecture the control plane is deployed as a distributed entity at each router, however this is contrary to SDN philosophy because both planes are placed inside the router. As far as the evaluation is concerned, the authors compared their SDNoC routing approach with the static and dynamic routing schemes in the traditional on-chip network. Through the results they showed that SDNoC is able to improve the network performance and reduce power consumption. Furthermore, the authors clearly stated that more details along with standardization between control and data plane, as provide by the OpenFlow specification, need to be considered in the future.

Afterwards, [Sandoval-Arechiga et al., 2015] applied SDN principles in order to propose an SDNoC architecture. This architecture focus on abstraction layers and interfaces that permit its deployment in a modular fashion by potentially helping to overcome the NoC management problems in the many core era. More precisely their architecture consists of three layers: Operating System, Network Operating System, and Infrastructure and five planes: Applications, Network Management, Control, Data Forwarding and Data Processing. However, the authors proposed an architecture with-

out providing enough details about the communication protocols between layers. Another interesting contribution of the same authors is presented in [Sandoval-Arechiga et al., 2016], where they evaluate the SDNoC architecture among PE in a many core system with System C simulator, focusing on the configuration time, delay, and throughput of their architecture. Precisely, the authors presented a system model of $7x10$ 2D mesh tiles, where each tile is composed of a SDN router, NI, memory and a PE. The controller was modeled in System C as a process running in the PE of tile at the center of the mesh, with the functionalities of sending packets for PE and routers configuration, stop and start computation in a per flow basis.

Thereafter [Scionti et al., 2016] proposed a scalable SDNoC architecture for future many core processors. Their design tried to merge the benefits of ring based NoC (i.e., performance, energy efficiency) with those brought by dynamic reconfiguration (i.e., adaptation, fault tolerance), while keeping the hard-wired topology (2D-mesh) fixed. Also, their interconnect architecture allows mapping different types of topologies and communication requirements. Specifically, their architecture consists of separate rings, which allow the communication flow in the north/south and east/west directions, while specific bits control the status of each link. Each PE has specific instructions to control the network topology by software, including switch off the links which are not used.The authors evaluated the proposed architecture on an in-house simulator, in order to test scalability and application latency, considering synthetic random traffic and a matrix multiplication kernel. Nonetheless, the existence of a controller has been neglecting along with the security issues of their architecture.

[Berestizshevsky et al., 2017] presented a detailed SDNoC architecture, based on a hybrid hardware/software approach. In their architecture they introduced a software based centralized Network Manager (NM), running on a dedicated core. The Network Manager (NM) allocates the routes and the routers forward the packets without storing them. Also, the routers do not maintain any routing tables. The authors evaluated the performance of the SDNoC scheme with a custom simulators. An improved solution is recently introduced by [Fathi and Kia, 2017] where all the routers do not need to reach the controller. The router attached to the source IP core sends the packet header to controller and controller provides a sequence of exit ports at each router on the route. All the other intermediate routers check the packet header and forwards the packet to already mentioned exit port. The proposed architecture has been tested within ISE Xilinx. Moreover, in the context of communication demands of future multi-core systems, [Zhou

and Zhu, 2017] proposed a Dynamic Task Mapping Algorithm (DTMA) for SDNoC, with the purpose of minimizing the communication cost of the application execution and achieving the load balance among routers. At the same year the author of [Salvador et al., 2017] proposed an SDNoC controller that permits run time reconfiguration of the data forwarding plane and at the same time allowing the execution of different algorithms in run time. Specifically, they presented a bus-based SDNoC controller, capable of generating the requested services by upward layers, for the reconfiguration of data forwarding and processing devices. However the authors did not provide any evaluation of their proposal. Thereafter, [Ruaro et al., 2017] propose a SDNoC Circuit Switching (CS) infrastructure for many-core systems. Their approach enabled the design of a simple Multi-Physical Network (MPN) for CS, through configurable CS routers based on elastic-Buffers. The main goal of their contribution was to establish CS for real-time applications flows by run-time support. Furthermore, with the help of a clock-cycle accurate RTL model the authors evaluated their approach.

[Ruaro et al., 2018] presented the pros and cons of the SDNoC paradigm based on their previous architecture. Precisely, they simulated a cycle-accurate many-core model, filling the lack in the literature by proposing a generic SDNoC architecture, addressing hardware and software implementation details. The authors compared the quality of the proposal with a state of the art search path mechanism (hardware implemented), in a QoS case-study providing CS for applications. The same year another interesting contribution has been published by [Scionti et al., 2018]. The authors, based on their previous contribution [Scionti et al., 2016] provide more information about their SDNoC architecture by targeting specific hardware, cloud to high-performance many-core processors in the cloud data centers. Also, they provided simulation results by evaluating the power consumption, area and performance of different SDNoC topologies, allowing local and global traffic to be decoupled.

Later on, [Silva et al., 2019] presented a communication latency evaluation of SDNoC. In their architecture the manager (or controller) is able to execute two routing algorithms in order to define a path from a source to a given destination, the XY deterministic routing and the Dijkstra adaptive algorithm. Also, they provided an evaluation of their architecture by comparing the two routing algorithm by using system C language. The same year another interesting contribution proposed by [Ruaro et al., 2019], who propose a novel distributed SDNoC architecture, with multiple controllers,

each of them is managing one cluster of routers. In this work also, the authors proposed a short path establishment heuristic for global paths that explores the controllers' parallelism. Finally the authors, compared their distributed SDNoC architecture with one of their previous proposed centralized SDNoC architecture [Ruaro et al., 2018] by concluding that their new architecture outperformed their old one in total latency in systems larger than 256 cores without loses in success rate.

Finally, in 2020 the security aspect of SDNoC start attracting more attention by researchers. In [Ruaro et al., 2020], the authors presented a systemic and secure SDN framework for NoC-based many-cores, allowing that only a trusted controller can define the communication path within a source and a destination router. Also, they described the iteration between the hardware, operating system, and user's tasks, provided a secure SDN router configuration protocol. Their architecture manages a Multiple-Physical NoC, with one packet-switching subnet and a set of circuit-switching subnets. As far as their experimental results, they manage to show the framework's capability to avoid DoS and spoofing attacks while presents a low router configuration overhead by comparing the performance of their protocol with [Soultana Ellinidou, 2019].

3.3.2 Discussion

The SDNoC communication paradigm attracted the attention of academia during the past years starting from 2014 [Cong et al., 2014]. However, the researchers explored different network architectures [Sandoval-Arechiga et al., 2015, Berestizshevsky et al., 2017, Ruaro et al., 2018], it can be noticed that the network entities were always the same (routers, controller or NM, Network Interface (NI), PE or core or IC and also the routing was handling by the controller, respecting the SDN concept. In each paper the authors focused on different aspects of SDNoC as for example in [Sandoval-Arechiga et al., 2016] the authors focused on the evaluation of SDNoC within a hardware simulator, in [Salvador et al., 2017] the authors focused on the functions and services of the controller and in [Scionti et al., 2018] focused in a ring topology and the reconfiguration aspect of SDNoC. Moreover, It has been observed that most of researchers focused on the hardware aspect of SDNoC by neglecting the network but also the security aspects of it. In the context of SDN technology, the OpenFlow [McKeown et al., 2008] protocol has become the de facto protocol for communication between controller and the routers, however OpenFlow protocol has been designed for large scale networks and its adoption into micro-scale networks is impossi-

ble. As far as the security is concerned, the [Ruaro et al., 2020] presented a systemic and secure SDN framework for NoC, however a security model for SDNoC but also the exploration of different attacks within SDNoC remained unexplored.

3.4 SDNoC Architecture

The main idea of SDNoC is to minimize router complexity by exporting the routing logic to a centralized controller which has a general view of the network and can take routing decisions efficiently. Precisely, a decrease of the complexity and area of the routers is achieved by the separation of the control logic which is implemented within hardware and the placement of it within a software-based network controller. Decoupling the control layer from the physical layer simplifies the router's design. Furthermore SDNoC provides better re-usability because routers are generic and simple hardware components, configured by software. The path between any communicating pair in the system requires the configuration of the routers belonging to the path. From an architecture point of view, the only difference between SDNoC and NoC is that the SDNoC manages routing in an adaptive manner with the help of a centralized controller (Figure 3.2).

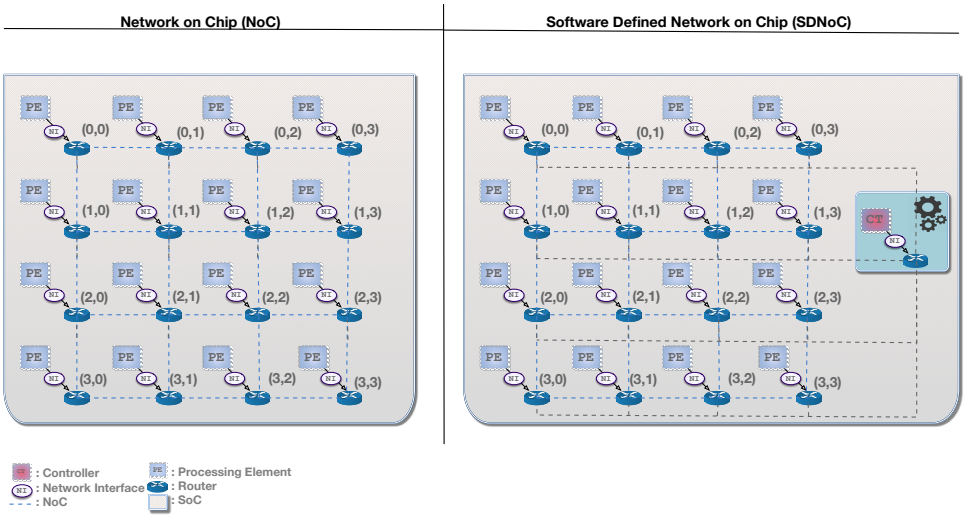


Figure 3.2: NoC vs SDNoC architecture.

According to the authors of [Cong et al., 2014, Ellinidou et al., 2019] SDNoC could possibly be adaptable for SoCs thanks to its advantages: 1) it reduces the hardware complexity, 2) it has high re-usability and 3) it has flexible management of communication policies. However, there are also some challenges that should be taken into account, in particular the high overhead for path selection in software against hardware based approaches and the centralized controller which can be a single point of failure.

An SDNoC architecture is depicted on Figure 3.3. The main entities of an SDNoC architecture are: Network Interface (NI), Physical Links (PL), Routers, Processing Element (PE) and the Controller. The routers are linked to every PE which could be a memory, a core or a processor and interconnects them through physical links. The NI is the intermediate entity between a PE and a router. More specifically, packets are traveling between different nodes of the network, routers, and the packet routing is managed by a centralized controller, which is running as a process on a given PE.

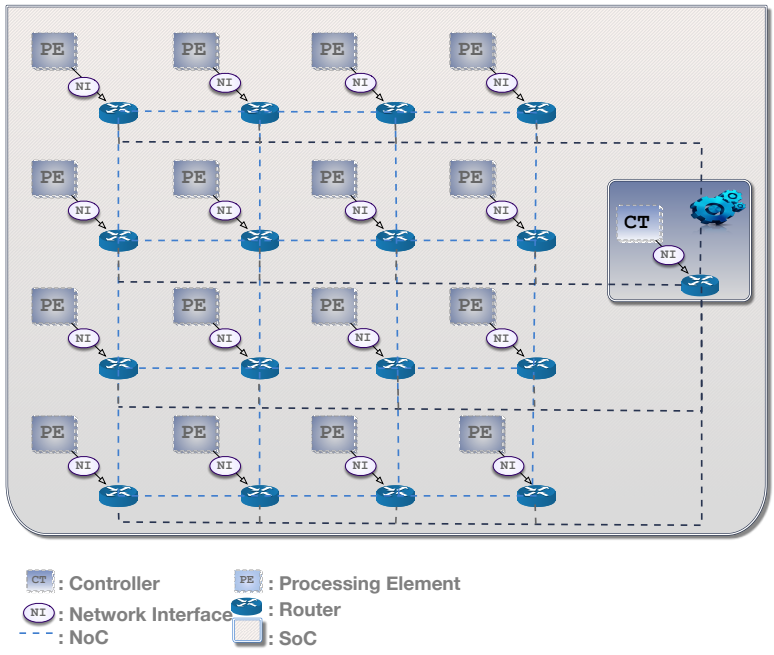


Figure 3.3: SDNoC architecture.

The network entities of an SDNoC architecture are explained below:

- **Routers**: Figure 3.4 illustrates the architecture of a SDNoC 5-port router employing VC flow control and SDN based switching. The five ports correspond to the four cardinal directions (North (N), West (W), South (S), East (E)) and the Local (L) direction which connects the router with the PE through NI. The router consists of five components: the Flow Tables, the VC Allocator, the Switch Allocator, the Buffers and the Crossbar Switch. The SDNoC router consists of a two-stage, pipelined architecture. The first stage is responsible for routing, where the router checks the Flow Tables, which they include flow entries with a *Match field* with the source and destination ID of a packet and an *Action field* with the outport direction for the flow entry. If there is not any flow rule for the given source and destination then the router will send a request to the controller to ask a new route. The second stage is responsible for crossbar traversal. In this work, the functionality of the router is described with respect to a 2D mesh interconnect.

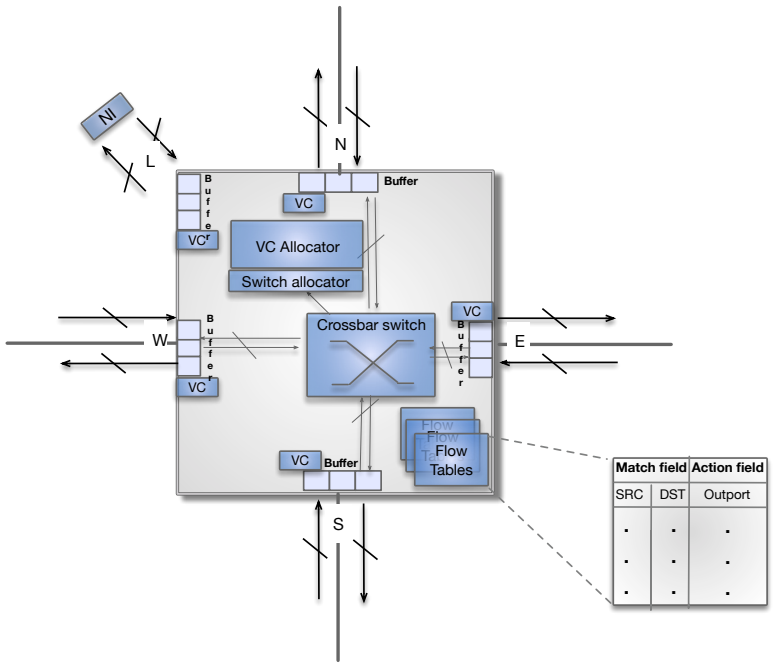


Figure 3.4: SDNoC router architecture.

- **Controller:** The SDN controller consists of a series of functions for

sending packets for PE and router configuration, stop and start computation in a per flow basis. The controller provides the following services: sending configuration to a specific set of nodes in the network; collecting state and statistics data from a specific set of nodes in the network and generating a global or partial view (state) of the network. This software-based control enables to reduce the hardware complexity, moving the decision to establish the network paths to the software.

- **Physical Links**: The communication between the controller and routers is managed by dedicated links, which interconnect them. Through the physical links the controller transfers control messages related to routing decisions to the routers and monitors the data network state.
- **Network Interface (NI)**: is composed by two FIFO memories, one logic block to interface with the network, called router adapter, and a logic block to interface with the processing unit (or core), called core adapter.

As far as the extra resources for the hardware implementation of the SDNoC and more precisely the controller block, an extra low power General Purpose Processor (GPP) of 8-32 bit will be needed [Schmidt et al., 1993]. A GPP is easily programmable by the user, it is designed for a variety of computation tasks and it can interact directly with all PE through the memory.

The SDNoC can accommodate larger topologies, however the trade-off for the communication router-controller will be a burden. For this reason, and by taking into account the future SoC architectures like CoC and chiplet, the proposed SDNoC architecture can be available for a cluster of cores or for the inter-chiplet communications. Hence, in case of a bigger topology there is a possibility of hierarchical controllers manage the routing within different clusters of cores or different chiplets.

3.5 Routing within SDNoC

As stated in Chapter 2, routing is a process of selecting a path from a source to a destination node within a network or between different networks. Routing within SDNoC is a completely different process compared to routing within NoC. In an SDNoC architecture the routing logic of a router has been deleted and moved to a centralized controller, who has a

general view of the network and can take routing decisions efficiently. The designer is able to employ any routing algorithm based on application requirement in order to deliver a certain packet from a source to a destination. In this research, 5 routing algorithms were chosen to be implemented and evaluated within an SDNoC architecture following a novel proposed routing algorithm (Table 3.1).

Table 3.1: Routing algorithms implemented within SDNoC

Routing Algorithm	Year	Reference	Features
`XY`	-	-	simple, deadlock and livelock free
`West First (WF)`	1992	[Glass and Ni, 1992]	based on turn model, livelock free
`North Last(NL)`	1992	[Glass and Ni, 1992]	based on turn model, livelock free
`Negative First (NF)`	1992	[Glass and Ni, 1992]	based on turn model, livelock free
`Odd Even (OE)`	2000	[Chiu, 2000]	based on turn model, deadlock free
`Modified OE (OESL)`	2019	[Ellinidou et al., 2019]	fully adaptive, based on SDNoC architecture, deadlock and livelock free

3.5.1 XY Routing

XY is a *dimensional order routing* in which the packets are first routed in the horizontal direction until they reach the column, where the destination is located and then, they are routed in the vertical direction until they reached the destination. Therefore, XY routing is deterministic and minimal. Deadlocks are avoided since four turns are prohibited (Figure 3.5(a)), and livelocks are avoided since this algorithm implies minimal routes. XY routing is typically used for its simplicity. However, in some scenarios, XY routing does not evenly distribute the traffic across the network since most of the traffic is concentrated on the middle of the network. Therefore, this area could be prone to congestion, which implies a decrease of the network performance. Moreover, XY being deterministic, it is impossible to avoid congested routers.

3.5.2 West First Routing

In West First (WF) routing, packets firstly move to the west as long as necessary and then they follow any path by avoiding turns to the west because they are forbidden. Therefore, any route can be taken if the destination is on the right-hand side of the source since no turn to the west is required. Otherwise, the routing is deterministic. Figure 3.5(c) shows the valid and invalid turns considered to prevent deadlocks.

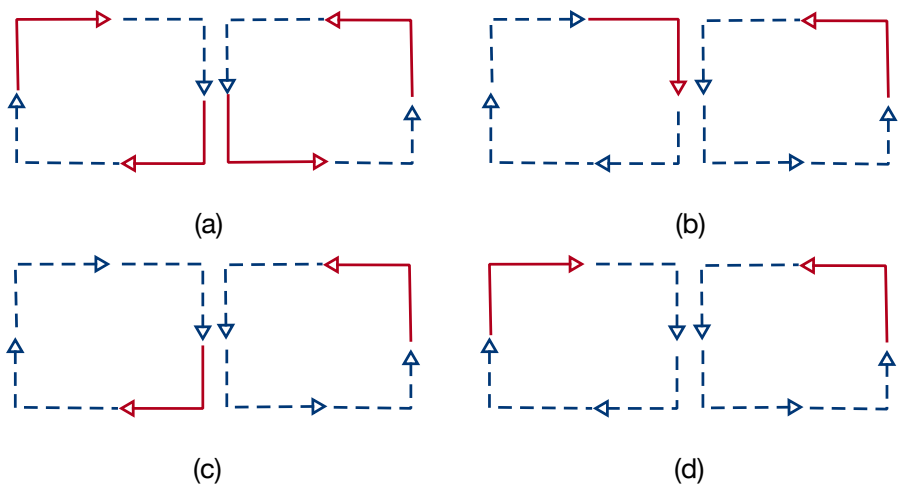


Figure 3.5: (a) XY. (b) Negative-First. (c) West-First. (d) North-Last.The solid red lines indicate the non-valid turns and the dashed lines indicated the valid turns.

3.5.3 North Last Routing

In North Last (NL) routing, packets are routed in any direction, but turns to north are performed at the end. In other words, once a packet flows across an output channel heading to the north, the following output channels, that will be used, cannot face the other directions. Therefore, any route can be taken if the destination is south of the source since no turn to the north is required. Figure 3.5(d) shows the valid and invalid turns in NL routing.

3.5.4 Negative First Routing

In Negative First (NF) routing, packets are firstly routed towards the negative directions, west and south, and no turn towards these directions is allowed later. Therefore, if the destination is north-east or south-west of the source, any minimal route can be taken. Otherwise, the routing is deterministic. Figure 3.5(b) shows the valid and invalid turns used to prevent deadlocks.

3.5.5 Odd Even Routing

The Odd Even (OE) routing was introduced in order to provide a more even degree of adaptiveness. In OE routing the columns of the mesh architecture are separated as odd or even. The first column is even, the second column is odd. The admissible routes have to obey the two following rules:

Rule 1: In an even column, a turn from the east to the north or the south is forbidden.

Rule 2: In an odd column, a turn to the west is forbidden.

These two rules ensure the deadlock-freedom of the OE routing algorithm. In Figure 3.6, the blue lines indicate the valid turns and the red lines indicate the non-valid turns.

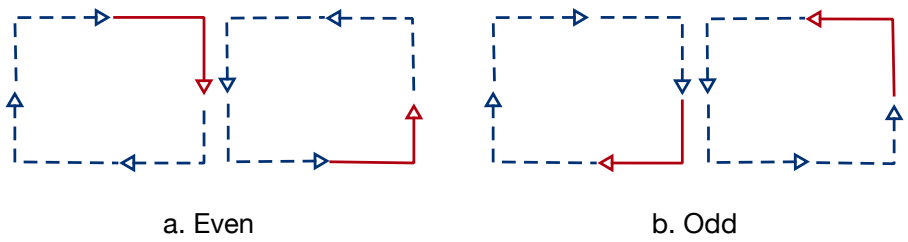


Figure 3.6: Odd-Even Routing

3.5.6 Modified Odd Even (OESL)

The proposed routing algorithm has two main functionalities: the computation of the admissible routes and the selection of a route among the admissible routes. In order to compute the admissible routes, the proposed routing algorithm relies on a turn model routing algorithm, OE [Chiu, 2000]. OE tends to provide better performance and higher adaptiveness than the other turn model algorithms. Based on the SDNoC architecture, once the controller has computed a set of admissible routes using the OE routing algorithm, it applies a selection function on the set in order to get the best possible route.

The *selection function* has a set of routes and the network state as inputs and outputs the optimal route from the set. In order to determine which route is the optimal one, the first step is to define metrics that assess the routes. The proposed selection algorithm that is implemented within the controller takes into account the link load and the router load. The load of a link (l_i) corresponds to the number of flits per second that flow through the link. The router load (r_{ij}) is the number of flits per second arriving towards the router. When the selection process takes place, the controller is responsible to run an algorithm selecting the best route from an admissible set of routes. For this reason the two aforementioned metrics were designed in order to avoid the highly-loaded links and routers within

the route.

Highly-loaded links affect the bandwidth and their corresponding input buffers are likely to be full. Therefore, sending packets towards highly-loaded links will imply a considerable latency for the incoming packets since they will have to wait for the release of the links and the corresponding input buffers before accessing them. On the other hand, the more a router is loaded, the more time it will take to process incoming packets since it has to process first the already present packets. By avoiding the highly-loaded links and routers, the selection function (SL_{sum}) aims to balance the traffic as much as possible across the data network and therefore avoids the formation of congested network areas. In order to determine the best route among an admissible routes set, these metrics have to be used to evaluate the routes. In this case, the controller computes a score (S) for each route among the set using a combination of the aforementioned metrics. With the proposed selection function, the route scores are computed by summing the load on the links and the routers along the routes. This score computation is computed with the following equation:

$$SL_{\mathrm{sum}} = \sum_{i=0}^{L_{\mathrm{f}}} l_i + \sum_{i=0}^{S_{\mathrm{f}}} \sum_{j=0}^{S_{\mathrm{f}}} r_{ij}. \tag{3.1}$$

Where L_f is the number of the sets of the link load values along the route and S_f the number of the sets of the router load values along the route. The controller is aware of the load of the links from the network monitoring process, and the load of a router is inferred from the load on the links arriving towards the routers as:

$$r_{ij} = \sum_{i=1} \frac{l_i}{L}. \tag{3.2}$$

Where r_{ij} is computed as the average load on the links arriving towards the router so that the router load and the link load stay in the same order of magnitude and L is the number of the router links. Thereby, the route score is equally affected by the load on the links and on the routers.

At the end, the controller computes the S for each route within the set according to the SL_{sum} and chooses the route with the lowest score. In the case of multiple routes having the same S, a random choice is made.

3.6 MicroLET Protocol

MicroLET is the first SDNoC-based communication protocol for chipLET-based systems [Ellinidou et al., 2019]. The design of MicroLET is based on the above mentioned SDNoC architecture. More precisely the SDNoC integration within chiplet-based systems is depicted in Figure 1.4 and described in Chapter 1.2. As it has previously been mentioned the Openflow [McKeown et al., 2008] is the most common and widely used SDN-based communication protocol, which enables communication between controller and routers. However, Openflow is designed for large scale networks and its integration to micro-scale networks seems impossible due to many network messages that needed to be exchanged between participants. Hence the necessity of a novel lightweight SDNoC communication protocol destined for future SoCs is obvious. Therefore, the MicroLET SDNoC-based communication protocol was introduced, which consists of 3 main phases: 1) Handshake Phase 2) Network Monitoring Phase 3) Routing Phase, which are detailed in Section 3.6.3.

3.6.1 Packet format

Processing cores exchange data among themselves by sending packets across the interconnect and consequently through routers. Furthermore a router sends packets to controller but also to the other routers by using the data link layer. A packet is divided into a sequence of fixed-length flits, which are composed of a header flit, body flits, and a tail flit. The packet format in the SDNoC is illustrated in Figure 3.7 and it includes 8 fields:

- **TYPE:** indicates the type of the messages (different type fields are shown in Table 3.2).
- **SRC:** consists of the source ID.
- **DST:** consists of the destination ID.
- **NEXT_HOP:** consists of the next hop ID.
- **PRIO:** contains the priority of the packet, which can be high or low in order to be pipelined accordingly.
- **PAYLOAD:** contains the real data.
- **TS:** is the timestamp and represents the send time.
- **CRC:** represents the Cyclic Redundancy Check, which is the error-detecting code field.

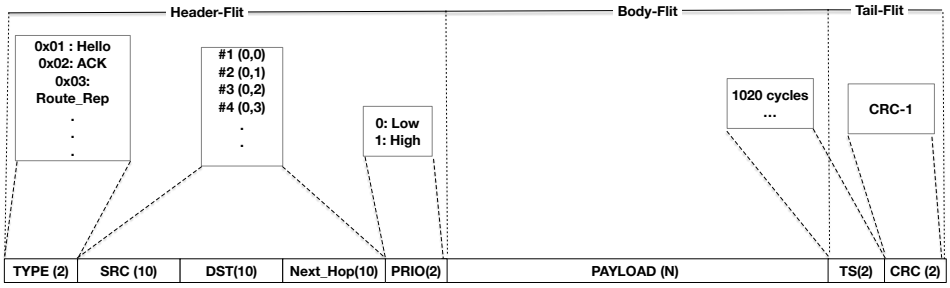


Figure 3.7: Packet format

3.6.2 Network Messages

The network messages are exchanged between the network entities through physical links. The different types of messages, which are integrated in order to fit in the packet format, are illustrated in Table 3.2. The communication protocol includes 8 types of messages with different content. The `HELLO` message is designed for the handshake phase and the `ROUTE_REQUEST, ROUTE_REPLY, FLOW_UPDATE, NET_REQUEST, NET_REPLY` are designed for the network monitor and routing phases. Furthermore, it is important to mention that every `ROUTE_REPLY, FLOW_UPDATE, NET_REPLY` message should be acknowledged by an `ACK` message, otherwise it should be retransmitted.

3.6.3 Communication Protocol Phases

The MicroLET communication Protocol consists of 3 main phases:

1. **Handshake Phase:** During the Handshake Phase a `HELLO` message is exchanged between the participants. In this way, the controller is aware of how many routers are in the network and about their ID's.

2. **Network Monitoring Phase:** In order to move to the Network Monitor phase, the Handshake phase should take place beforehand. The controller requests to be informed about the network state by periodically (within a period τ, explained in Chapter 4.3.1.1) sending a `NET_REQ` message to the routers. The receiver router should reply with a `NET_REPLY` message, which includes the current flits passing by every port. Each router has a counter in the buffer of every port and it is increasing according to the flits that are coming from this port in a given period. Therefore, each router monitors the flits that

are inserted through the North, East, South, West and Local ports during an interval time and forms the `NET_REPLY` message. As soon as the controller receives a `NET_REPLY`, it should send an `ACK` back to the routers and it updates its parameters that would be needed for the next phase. With this process the controller manages to have a network state view, which is the key element for the *selection function* during the routing phase.

Table 3.2: Designed Network messages

Type	T-Value	Description	Contents
`HELLO`	0x01	Sent by router to controller and vice versa or by router to router for the handshake process.	HELLO
`ACK`	0x02	Sent by router to controller or by transmitter router to receiver router to acknowledge the request.	ACK
`ROUTE_REQ`	0x03	Sent by router to controller which asks a route for a packet.	Packet ID
`ROUTE_REPLY`	0x04	Sent by controller as an answer to a route request message.	Packet ID, Route
`FLOW_UPDATE`	0x05	Sent by controller to routers in order to update the outport of a packet.	Packet ID, Route
`NET_REQ`	0x06	Sent by controller to routers which asks information for network.	NET-REQ
`NET_REPLY`	0x07	Sent by routers as an answer to a network state request message.	N= #, S=#, E=#, W=#, L=#
`DATA`	0x8	Contains the data.	PAYLOAD

3. **Routing Phase:** When the controller receives a `ROUTE_REQ` message from a source router, it extracts the `PACKET_ID` and the `SRC` and `DST` addresses from the upcoming flits which will be the input of the routing algorithm function. Afterwards, based on the source and the destination, the routing algorithm outputs a set of admissible routes. Therefore, the routing algorithm has two main functionalities: (1)the computation of the admissible routes and (2)the selection of a route among the admissible routes. In order to compute the admissible route sets, the proposed routing algorithm relies on a turn model routing algorithm. These algorithms have the advantages to

be lightweight and deadlock-free. Among the existing turn model routing algorithms, Odd Even [Chiu, 2000] is used since it tends to provide better performance and higher adaptiveness than the other routing algorithms. Finally, once the controller has computed a set of admissible routes using the OE routing algorithm, it applies the selection function on the set in order to get the best possible route and forms the `ROUTE_REPLY` message. The modified version of OE routing algorithm is described in Section 3.5.

3.7 Summary-Discussion

In this chapter, the SDN technology was presented together with its security issues and the SDN communication protocol OpenFlow. Afterwards the different research contributions that have been already seen in literature were described. It is followed a discussion about how the SDN technology will solve the challenges of the classic NoC but also about the new challenges that SDNoC need to encounter. In each research paper, seen in literature, different SDNoC architecture was described.

As it previously mentioned the SDNoC is introduced in 2014 by [Cong et al., 2014], however they were multiple contributions during the last years, most of researcher explored the hardware implementation of it by neglecting the networking and security aspect of it. Hence in this thesis, an effort has been made in order to address this two main field within SDNoC field. Furthermore each researcher individual described and different SDNoC architecture, hence in this research a potential prototype by respecting both the original SDN architecture but also the hardware integration of the SDNoC is presented together with a novel communication SDNoC communication protocol.

As far as the security is concerned, the [Ruaro et al., 2020] presented a systemic and secure SDN framework for NoC. In their approach the authors considered an architecture specific threat model based on malicious packet software task able to perform: DoS: generation of an incorrect SDN-router configuration packet aiming to crash the controllers' NI, flooding: flood the controllers' NI with malicious SDN-router configuration packets and spoofing: malicious packet trying to assume the identity of one actor of the framework (global manager, controller, manager). It has to be mentioned that in their approach they consider both a controller but also a global manager for managing the configuration and routing of the network in comparison with the proposed approach where only a centralized controller

is used. However, a security model for SDNoC but also the exploration of different specific attacks within SDNoC remained unexplored. Hence, the second priority of this thesis was the exploration of the security within SDNoC, by firstly proposing a secure Group Key Agreement (GKA) communication protocol in order to ensure not only the secure configuration of the routers through the controller but also the secure communication between routers (Chapter 5.2). Afterwards the possibility of a specific attack, in this case a novel Hardware Trojan (HT)-Greyhole attack within SDNoC is explored for first time (Chapter 5.4). Furthermore the Byzantine faults, which are arbitrary faults within SDNoC, are investigated and a novel algorithm for the controller in order to tolerate the Byzantine faults is designed.

Moreover different routing algorithms and a novel routing algorithm based on an already existing one were presented in the context of SDNoC. The routing within SDNoC is a key element due to flexibility of defining and choosing different paths for a packet that a controller can apply to the routers of the network. Hence, the controller can choose one or multiple routing algorithms in order to find the best path for a given source to a given destination. Additionally, a novel communication protocol specifically designed for microscale networks, called MicroLET was introduced along with a novel packet format and a new message stack. Finally a novel routing algorithm based on an already existing one was presented in the context of SDNoC. The evaluation of the SDNoC architecture with the above mentioned routing algorithms will be presented on Chapter 4.

Chapter 4

Implementation and Evaluation of SDNoC

4.1 Introduction

Following Chapter 3, in this chapter an implementation and evaluation of the proposed Software Defined Network-on-Chip (SDNoC) prototype is described. Firstly, the different Network-on-Chip (NoC) simulators together with the chosen simulator are presented in detail. Afterwards, the implementation of the proposed SDNoC architecture is described, followed by the changes that have been made within the chosen simulator. Thereafter, an evaluation of the first SDNoC communication protocol (MicroLET), which was introduced in Chapter 3 is presented. The protocol is designed in order to provide a new routing approach based on Software Defined Network (SDN) technology and a new message stack specifically designed for microscale networks. It follows a performance evaluation of different routing algorithms under different scenarios. The considered scenarios consist of 3 topologies, under 3 different traffic models and under multiple injection rates. Since the scenarios are based on numerous sources of randomness the distribution converges to a normal or Gaussian distribution. Hence, the standard deviation coverage of the different scenarios is analyzed. Another research question that is answered within this chapter is how the performance is affected by the different parameters, for this reason a statistical analysis is followed in order to show the interaction of the different parameters within the network. Precisely an ANalysis Of VAriances (ANOVA) between latency and throughput and each factor separately: traffic injection rate (tir), routing, and traffic is performed.

Table 4.1: NoC Simulators

Category	Simulator	Language	Topology	Routing	Traffic
Regular network	NS-2 [Hegedûs et al., 2005]	C++, Otcl	Mesh	Dynamic	Constant
	OMNeT++ [Al-Badi et al., 2009]	C++	Torus	Dimension-order	Exponential
NoC dedicated	Nostrum [Lu et al., 2005]	System C and Python	Mesh, Torus, Ring, etc	XY, Deflection	Synthetic
	Noxim [Catania et al., 2015]	System C	Mesh	multiple	Synthetic
	Nirgan [Jain et al., 2007]	System C	Mesh, Torus, Ring	Source, XY and OE	Synthetic, embedded
	[Grecu et al., 2008]	Java	multiple	user-defined	Uniform
	Booksim [Jiang et al., 2013]	C++	multiple	multiple	Synthetic
Full-system	Garnet [Agarwal et al., 2009]	C++	multiple	XY, Turn model and Random	Synthetic

4.2 NoC Simulators

In order to evaluate the proposed SDNoC architecture a NoC simulator should be chosen. A common challenge when selecting the right NoC simulator is that available tools usually are strong in certain aspects but they are having deficits in others. NoC simulators can be divided into 3 categories:

- **Regular network**: simulators that are used in communication network, like NS-2 [Issariyakul and Hossain, 2009], OMNeT++ [Varga, 2010]. These simulators utilize the similarities that exist between general networks and NoC.

- **NoC dedicated**: implement a high-level representation of NoC, modeling data at message level, providing a set of architectures and protocols, and evaluating the network with the chosen traffic pattern in terms of latency, throughput and power. Some of the most famous simulators are Nostrum [Lu et al., 2005], Noxim [Catania et al., 2015], Nirgam [Jain et al., 2007] and Booksim [Jiang et al., 2013].

- **Full-system**: integrate a NoC model into a full-system simulator. Garnet [Agarwal et al., 2009] is a NoC model integrated into gem5 [Binkert et al., 2011], which is a simulation platform for Chip Multi-Processor (CMP).

An overview of NoC simulators can be found in Table 4.1, where the programming language, the supported topologies, routing and traffic of each simulator are presented. All NoC simulators have their advantages and limits. Hence, in order to chose a simulator for the design and evaluation of the SDNoC prototype some parameters were defined. Precisely, the following parameters are of high importance in the selection of the best NoC

simulator: measurements options, routing options, routing settings, traffic options, configuration option, full-system simulation (for future work). Based on the above mentioned parameters, the Garnet simulator was chosen. Garnet enables the evaluation of system-level optimization techniques within a state-of-the-art interconnection network by obtain correct results. Furthermore, it supports the evaluation of novel network proposals in a full-system fashion. Also, it enables the implementation and evaluation of techniques that simultaneously use the interconnection network as well as other top-level system components, like caches, memory controller, etc. Such techniques are difficult to evaluate faithfully without a full-system simulator that models the interconnection network as well as other components in detail.

In particular, the Garnet2.0 version of the Garnet simulator, which provides 1-stage pipeline and more configurability for the users over the previous version, was used. Precisely, it provides a cycle accurate microarchitectural implementation of an on-chip network router. It leverages the topology and routing infrastructure provided by gem5's ruby memory system model. The default router is a state of the art 1 cycle pipeline. Each router in the topology file can be given an independent latency, which overrides the default. In addition, each link has 2 optional parameters: `src_outport` and `dst_inport` which are strings with names of the output and input ports of the source and destination routers for each link [Agarwal et al., 2009]. These can be used inside Garnet2.0 to implement custom routing algorithms.

4.3 Implementation of SDNoC prototype

In order to design and evaluate the proposed SDNoC architecture but also to evaluate and compare the performance of the different routing algorithms within the architecture, simulations were performed with the Garnet2.0 [Agarwal et al., 2009], which was build within gem5 simulator. The gem5 simulator was build within Ubuntu 18.04. The traffic within the simulator is generated by the processing cores according to the traffic injection rate, which is the average number of packets injected by the cores into the network per clock cycle ($0 < tir \leq 1$). Each core generates packets following a Bernoulli distribution, described in the specification of the simulator [Binkert, 2020], with mean tir. In other words, each processing core will indeed generate a packet each $1/tir$ clock cycles on average, but the actual time at which the packets are transmitted is random.

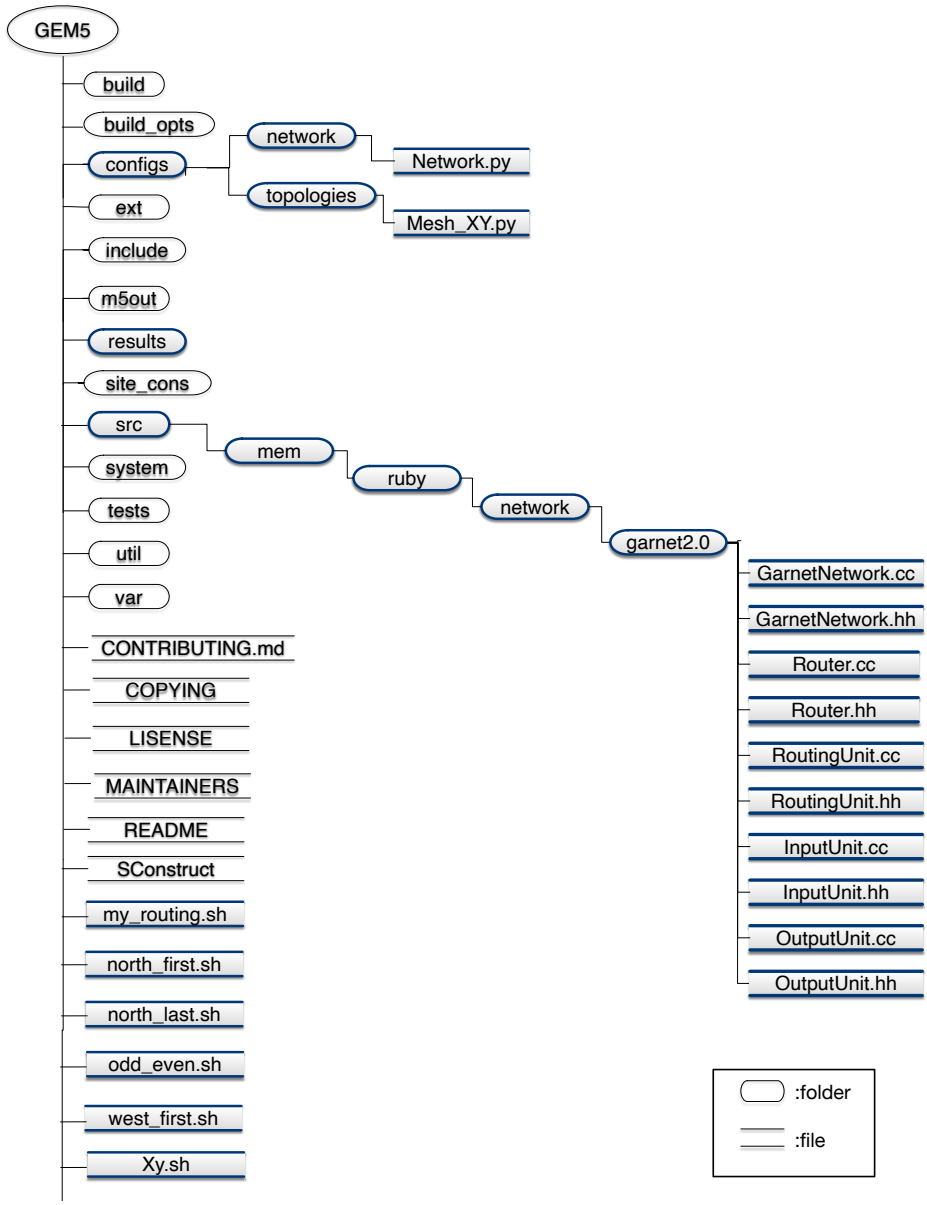


Figure 4.1: Modified and Added files tree

A folder tree of the modified and added files and folders within the official gem5 simulator is depicted in Figure 4.1. The main changes within the Garnet2.0 model has been made in the files: `GarnetNetwork.cc`, `Router.cc`, `RoutingUnit.cc`, `InputUnit.cc`, `OutputUnit.cc`, the code of which can be found on Appendix A. Specifically, an additional router is added that is linked with the controller, for convenience reason the code of the controller is implemented as a process within the router code. Hence the file `Mesh_XY.py` (Appendix A.7) is modified (main changes: lines `143-173`). Afterwards, in the file `GarnetNetwork.cc` (Appendix A.3) some extra lines of code were added (lines: `72-93`), in order to implement the SDNoC controller together with its NI. The main changes have been made in the file `Router.cc` (Appendix A.2), where links were created between all routers and controller and the routing process was modified (lines: `147-231`). Another file that was modified is the `RoutingUnit.cc` (Appendix A.1), where the controller computes the routes according to the selected routing algorithm and selects a route according to the selection function (lines: `238-345`). Also, the controller updates the flow tables of the routers along the selected route (lines: `345-364`) by following all the implemented routing algorithms (XY, WF, OE, NL, NF, OESL) in lines: `400-2043` . In the files `InputUnit.cc` (A.4), the main changes happened between lines `100-173`. Precisely, in the lines `103-140` it is checked if a flow entry exists for an incoming packet. If it doesn't exist (lines `117-122`), the packet is sent to the controller otherwise the packet is forwarded according to the router flow table (lines `125-139`). Furthermore, in order to test the SDNoC network under different parameters, 6 scripts were created that represent the different 6 Routing algorithms that are tested: `my_routing.sh, north_last.sh, north_first.sh, odd_even.sh, west_first.sh, xy.sh`. The results of the scripts are on the folder `results` (Figure 4.1).

4.3.1 SDNoC Parameters

As far as the performance measurements that have been used in order to evaluate the routing algorithms:

- **Latency**: the time that required to transmit a packet from a given source to a given destination. (clock cycles)
- **Throughput**: the number of the received packets by unit of time. More precisely the formula that has been used is the following:

$$Throughput = \frac{\sum received\ flits}{number\ of\ nodes * total\ cycles} \tag{4.1}$$

A majority of network studies require an interconnect model to be evaluated with synthetic traffic types as inputs. Such studies are very common in the interconnection network research community. Historically, several of synthetic traffic patterns are based on communication patterns that arise in particular applications [Bahn and Bagherzadeh, 2008]. Synthetic traffic stresses various network resources and provides an estimate of the network's performance under various scenarios. Furthermore Garnet has been designed to run in a network-only mode and supports only synthetic traffic types. Hence, in this scenario, three synthetic traffic patterns has been chosen: Transpose, BitReverse, Uniform, which are the most used synthetic traffics within NoC [Ma et al., 2014]. Two of them are shown in Figure 4.2:

Transpose: a node (i, j) only sends packets to its symmetric node $(n - 1 - j, n - 1 - i)$, where n is the size of the mesh.

BitReverse: Under BitReverse traffic, a source node sends packets to the node whose the address is the bit reversal of the sender address. For instance, a source node with the binary address $(b3, b2, b1, b0)$ sends packet to the node with binary address $(b0, b1, b2, b3)$.

Uniform: each node randomly sends packets to any other node with the same probability.

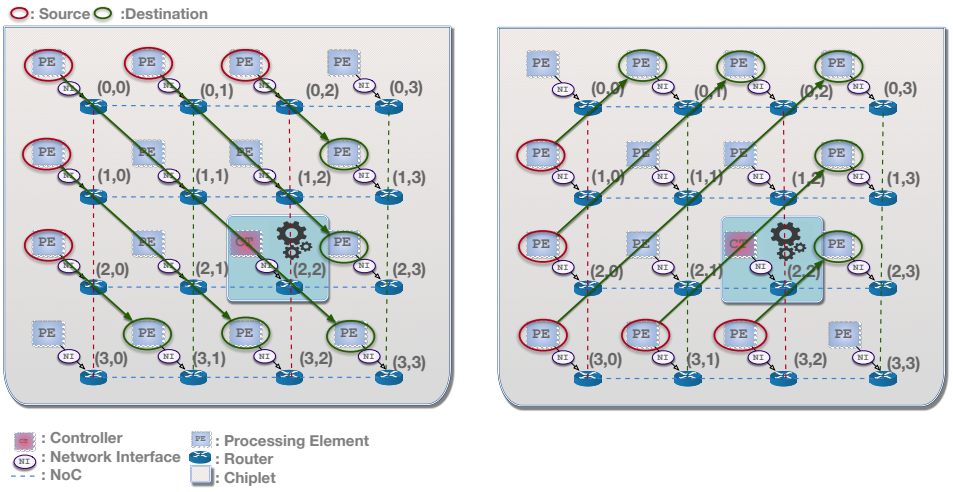


Figure 4.2: Source and destination under Transpose and BitReverse traffic

As far as the choice of the traffic injection rate, at low traffic loads, the average packet latency exhibits a weak dependence on the tir. However,

when the traffic injection rate exceeds a critical value, the packet delivery time rise abruptly and the network throughput starts collapsing [Ogras and Marculescu, 2005]. This is obvious from the Figure 4.3, where the average latency of different routing algorithms is presented under Transpose traffic and under multiple traffic injection rates. Hence by observing the performance of the routing algorithms under different injections rates, the traffic injection rate values of $0.015, 0.016, 0.017, 0.018, 0.019, 0.02, 0.022, 0.023, 0.024$ packet/cycle/core were chosen for the simulations. According to [Bahn and Bagherzadeh, 2008] similar average traffic injection rate is used also in the following applications: fft, radix, water-nsquared, water-spatial. Moreover from the Figure 4.3 it is obvious that the proposed OESL has the lowest latency during higher injection rate.

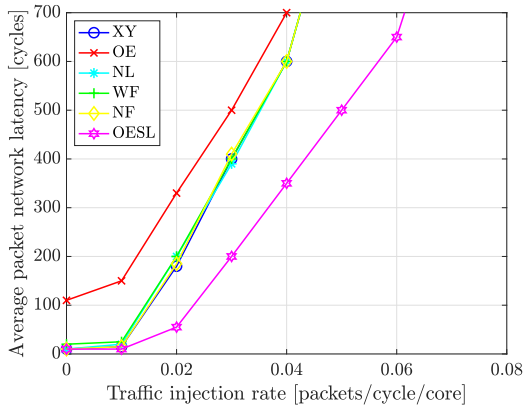


Figure 4.3: Average latency of Transpose traffic under different traffic injection rates (Topology:8x8).

4.3.1.1 Impact of τ

The evolution of the average latency and throughput as a function of the parameter τ for each traffic scenario is illustrated in Figure 4.4 [1]. Despite the traffic scenario the network performance is expected to decrease as τ increase since the monitoring process reports an older value of the link load. The routing does not rely on a correct view of the network state, which implies a network performance decrease. This case is observed under Transpose and Bit-Reverse traffic, due to the similar behavior of the traffic over multiple consecutive periods τ. Thereby, through the monitor process

[1] This work has been done under collaboration with Ir. Adil Layach during his Master thesis "Software-defined routing protocols for system-on-chip architectures".

an accurate view of the network state is reported to the controller and it is able to correctly answer a `ROUTE_REQUEST` message by calculating routes that avoid links and routers used by already deployed routes. For higher values of τ, the network performance is not affected because after a certain time all the possible source-destination pairs have already a certain route.

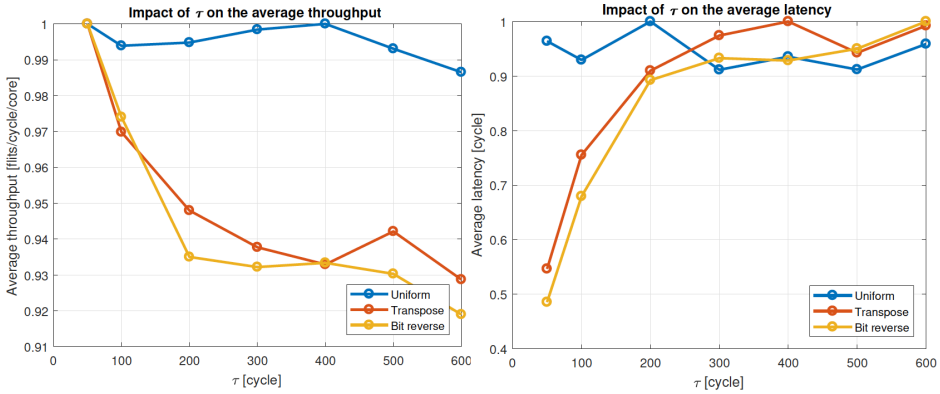


Figure 4.4: Impact of τ on the average latency and throughput (tir = 0.02).

On the other hand τ does not affect the network performance under Uniform traffic. This is expected due to the unreliability of this traffic scenario. Since the traffic is random, it is very likely to have different behavior from a period τ to another despite the actual value of τ.

4.3.2 MicroLET

In order to implement the MicroLET communication protocol within the Garnet2.0 simulator, based on SDNoC prototype, the 3 phases of the protocol (Section 3.6.3): Handshake Phase, Network Monitoring Phase, Routing Phase need to be taken into account. For the implemented scenario it is assumed that the Handshake Phase has already taken place. Concerning the Network Monitoring Phase, the `NET_REQUEST` and `NET_REPLY` messages were modeled as 1-flit packets. Nonetheless, they do not contain the content discussed before, because Garnet2.0 does not support the modulation of real payload within the exchanged packets. Moreover, regarding the Routing phase, in order to measure the link load, each router has a counter for each of its input channels. Each time a flit reaches an input channel, the corresponding counter is incremented. When the controller receives a `NET_REPLY` message from a router, it reads the value of the counters in order to get the link load and stores it within the corresponding $N \times N$

matrix. The routers manage the incoming packets according to flow tables. Also, flow table lookups are done in 1 clock cycle for the proposed network, however this might differ according the size of the flow table and the size of the network. When a router receives a packet that does not match with one of its flow entries, it forwards the packet towards the controller. Afterwards, the controller runs the routing algorithm in 1 clock cycle. Therefore, the time needed by the controller to compute the routes is not modeled. When the controller has computed a route, it updates the flow tables of the routers along the route and sends back the packet to the source router. The results and the routing algorithms are presented in the next section.

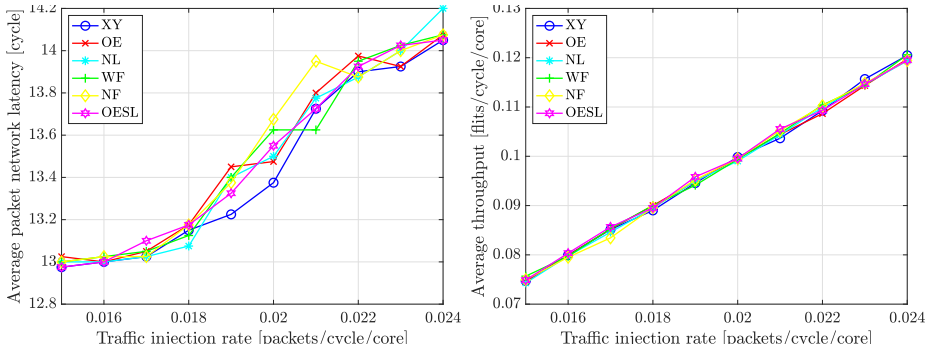
4.4 Routing Algorithms

The main focus was the investigation of the perfomance of different routing algorithms by evaluating their throughput and latency under different traffic scenarios and with a different number of nodes. Specifically, 3 topologies have been simulated: 2×2, 4×4, 8×8[2], under 3 different synthetic traffic scenarios: Uniform, BitReverse and Transpose (Figure 4.2), different traffic injection rates: $0.015, 0.016, 0.017,\ 0.018, 0.019, 0.02, 0.022, 0.023, 0.024$ packets/cycle/core, using different routing algorithms: XY, OE, NL, WF, NF and OESL. Also 40 iterations of each scenario were performed, according to convergence of the average value, and the mean value is depicted on the figures below. In contrast to the state of the art, in this research the performance of different routing algorithms within SDNoC under different synthetic traffic pattern scenarios and topologies has been investigated by bringing into the surface new scientific results about the performance of SDNoC and its possibilities to accommodate any kind of routing algorithm according to the traffic pattern in this case.

The simulation results have been categorized according to the different traffic patterns. In Figures 4.5, 4.6 and 4.7, the performance measurements (throughput and latency) of the 6 routing algorithms are depicted under Uniform traffic. In the scenarios of a 2×2 topology (Figures 4.5, 4.8 and 4.11) the throughput and latency and in the scenarios of a 4×4 topology (Figures 4.6(b) 4.9(b) 4.12(b)) the throughput measurements of the different routing algorithms are identical. Under Uniform traffic, in the

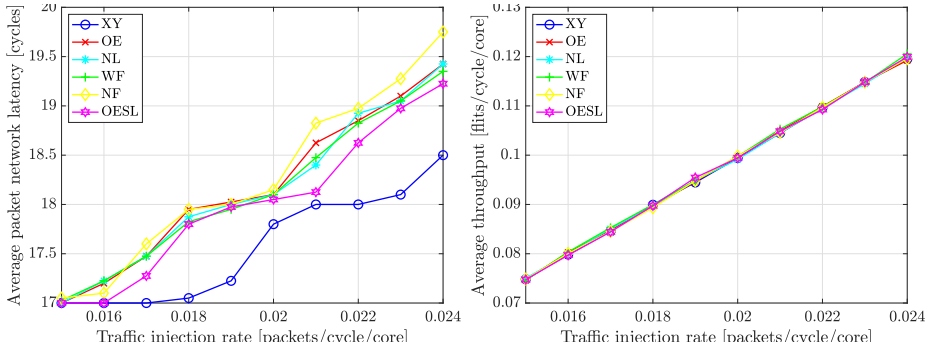
[2]Due to the limitations of the simulator no larger topology has been evaluated. However, as it previously mentioned the SDNoC is able to accommodate larger topologies by simulating cluster of cores and managing the routing within these clusters with the help of a cluster controller. Hence, by taking into account the future SoC architectures here a cluster of cores or a inter chiplet communication paradigm is simulated.

4×4 latency graph (Figures 4.6(a)) the XY routing algorithm outperform the others algorithms. Furthermore, in a scenario of an 8×8 topology (Figure 4.7) it is obvious that XY routing algorithm has the best performance (the lowest latency, the highest throughput) followed by the proposed OESL routing algorithm.



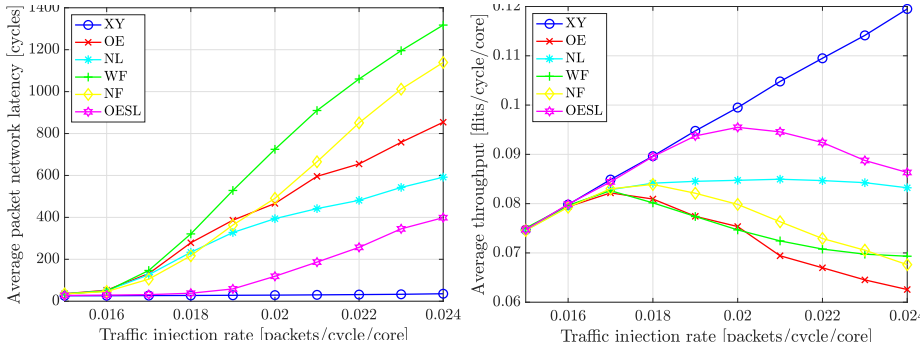
(a) latency **(b)** throughput

Figure 4.5: Performance measurements under Uniform traffic (Topology: 2x2 Mesh).



(a) latency **(b)** throughput

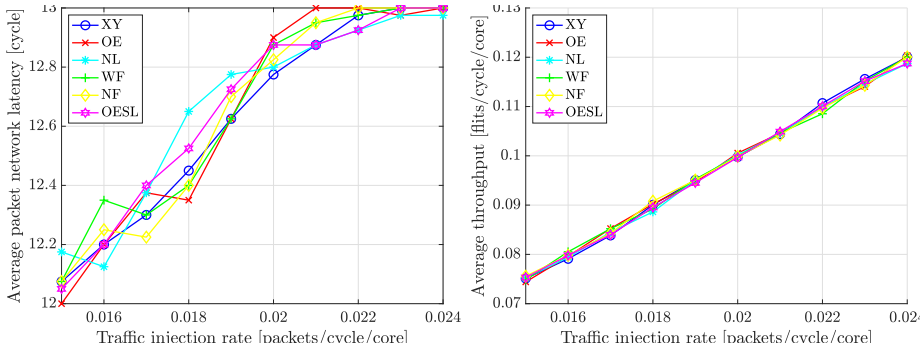
Figure 4.6: Performance measurements under Uniform traffic (Topology: 4x4 Mesh).



(a) latency **(b)** throughput

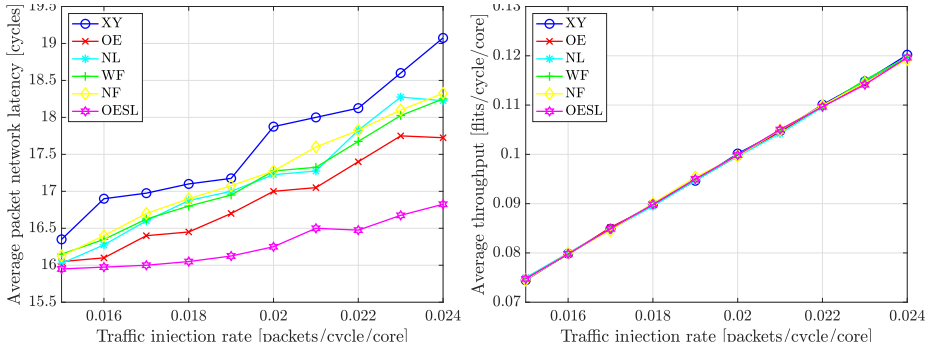
Figure 4.7: Performance measurements under Uniform traffic (Topology: 8x8 Mesh).

Figures 4.8, 4.9 and 4.10 depict the performance measurements (throughput and latency) of the 6 routing algorithms under BitReverse traffic. In 2×2 and 4×4 NoC topologies both latency and throughput measurements of the different routing algorithms are similar, except for the latency of the 4×4 Mesh topology (Figure 4.9(a)), where the proposed OESL has lower latency in contrast to the other routing algorithms. In a 8×8 NoC topology, OESL routing algorithm has the best performance.



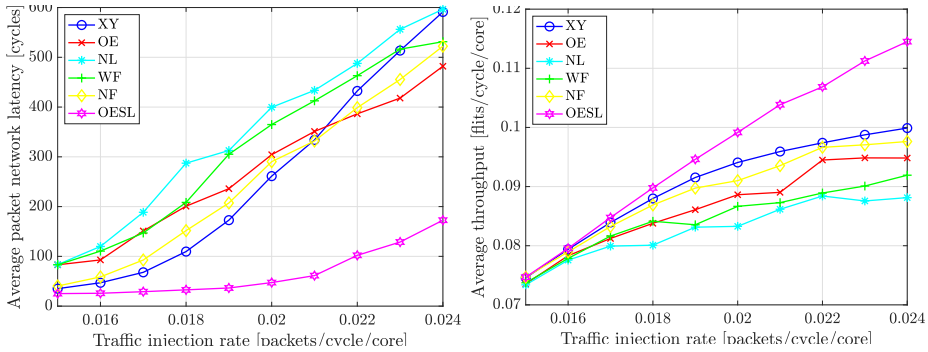
(a) latency **(b)** throughput

Figure 4.8: Performance measurements under BitReverse traffic (Topology: 2x2 Mesh)



(a) latency **(b)** throughput

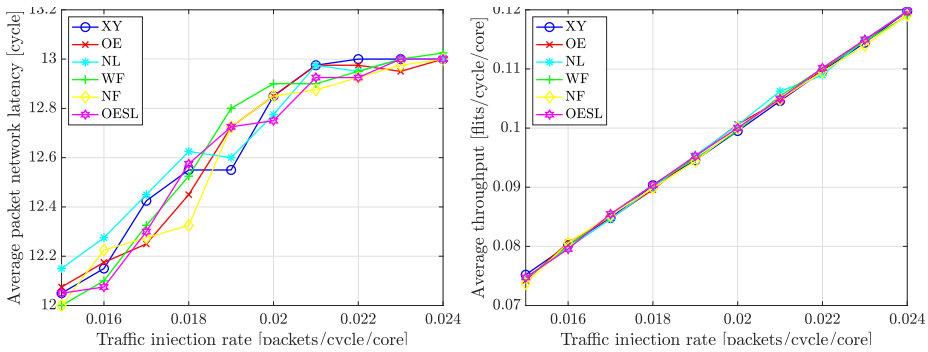
Figure 4.9: Performance measurements under BitReverse traffic (Topology: 4x4 Mesh)



(a) latency **(b)** throughput

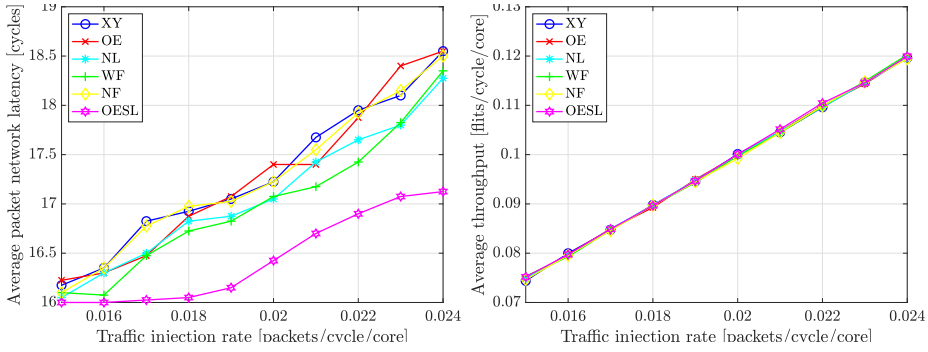
Figure 4.10: Performance measurements under BitReverse traffic (Topology: 8x8 Mesh)

Figures 4.11,4.12 and 4.13 depict the performance measurements (throughput and latency) of the 6 routing algorithms under Transpose traffic. In the 2×2 and 4×4 topologies, both latency and throughput measurements of the different routing algorithms are similar, except for the latency of the 4×4 Mesh topology(Figure 4.9(a)), where the proposed OESL has lower latency in contrast to the other routing algorithms. In the 8×8 topology, OESL routing algorithm has the best performance. Furthermore it should be noted that OE routing algorithm has the highest latency and lowest throughput, but with the help of the proposed novel selection function within OE, it achieves the lowest latency and the highest throughput. Furthermore, the proposed OESL algorithm outperforms the rest of the algorithms.



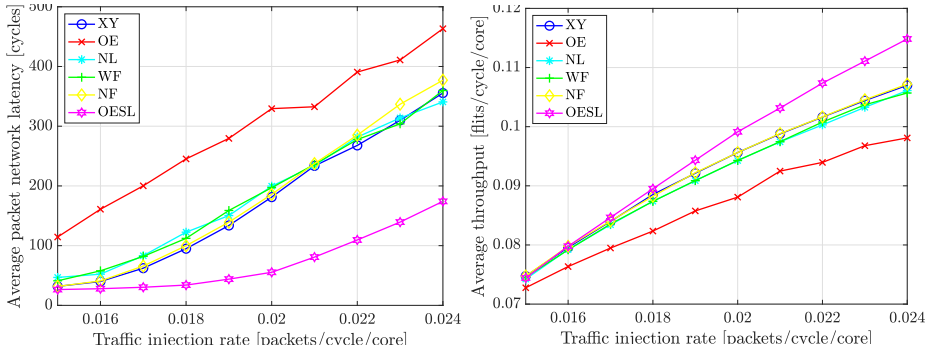
(a) latency **(b)** throughput

Figure 4.11: Performance measurements under Transpose traffic (Topology: 2x2 Mesh)



(a) latency **(b)** throughput

Figure 4.12: Performance measurements under Transpose traffic (Topology: 4x4 Mesh)



(a) latency **(b)** throughput

Figure 4.13: Performance measurements under Transpose traffic (Topology: 8x8 Mesh)

4.4.1 Standard Deviation Coverage

Since the proposed scenarios are based on numerous sources of randomness the distribution converges to a normal or Gaussian distribution. A normal distribution, called Gaussian or bell curve, is a very common continuous probability distribution in statistics but also in science. The normal distribution model derives from the Central Limit Theorem [Rosenblatt, 1956]. This theory states that averages calculated from independent, identically distributed random variables have approximately normal distributions, regardless of the type of distribution from which the variables are sampled. The Probability Density Function (PDF) of a normal distribution is:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.2}$$

, where μ is defined as the mean of the distribution, σ is the standard deviation and σ^2 defined as the variance.

The standard deviation (σ) is a measure of the amount of variation of a set of values [Bland and Altman, 1996]. A low standard deviation indicates that the values tend to be close to the mean of the set, while a high standard deviation indicates that the values are spread out over a wider range. The standard deviation follows the empirical rule, in which:

- 68% of data falls within the first standard deviation from the mean.
- 95% of data falls within two standard deviations.
- 99.7% of data falls within three standard deviations.

The graphs in the previous section represent the average value of latency and throughput of different routing algorithms under different traffics and different traffic injection rate. However, for a better understanding and since the proposed scenarios converge to a normal distribution the calculation of standard deviation and coverage was mandatory. Hence, for every 40 iterations of every scenario the 95% coverage of data that falls within two standard deviations ($\mu - 2\sigma, \mu + 2\sigma$) is determined, consequently the upper and lower bound of each mean were calculated. The results are depicted in the Figures 4.14 -4.22.

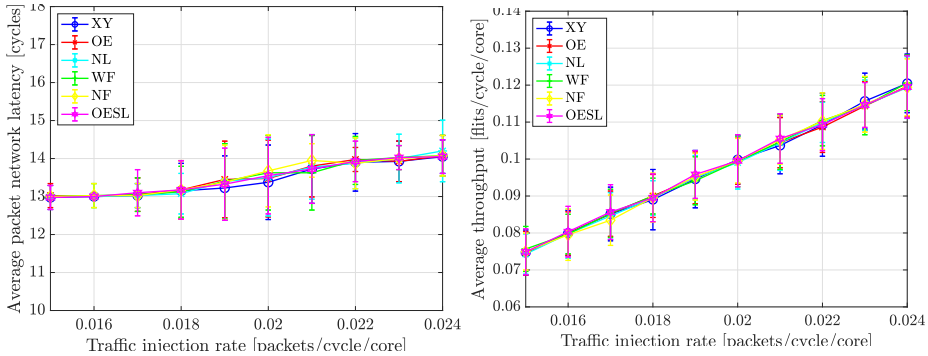
Precisely, in the 2×2 topology of Uniform, Transpose and BitReverse traffic it is obvious that the different values of the latency and throughput tend to a linear behavior (Figure 4.14, 4.17 and 4.20). Similarly in the 4×4 graphs of throughput of the different routing algorithms tend to a linear

behavior. However, in the latency graphs of 4×4 topology, the first difference between the routing algorithms could be noticed . In Figure 4.15(a) the latency values are between $14 - 15.5$ cycles. The XY routing algorithm has the lowest latency between $14 - 15.5$ cycles. Nevertheless, the difference between latency measurements is less significant in most of 4×4 topologies (Figure 4.18, Figure 4.20).

On the other hand, when a bigger topology is simulated (8×8), the differences between both latency and throughput graphs from the routing algorithms are more obvious. In the Uniform traffic scenario, the average packet latency (Figure 4.16(a)) was between $20 - 150$ cycles and the average throughput between $0.05 - 0.13$ flits/cycle/core (Figure 4.16(b)). More precisely, in Figure 4.16(a) the XY routing is linear in contrast to the other routing algorithms. However, with a traffic injection rate of $0.015 - 0.017$, the average latency of OESL and XY is identical, in higher traffic injection rate the latency of OESL is increasing in a similar way as the other routing algorithms.

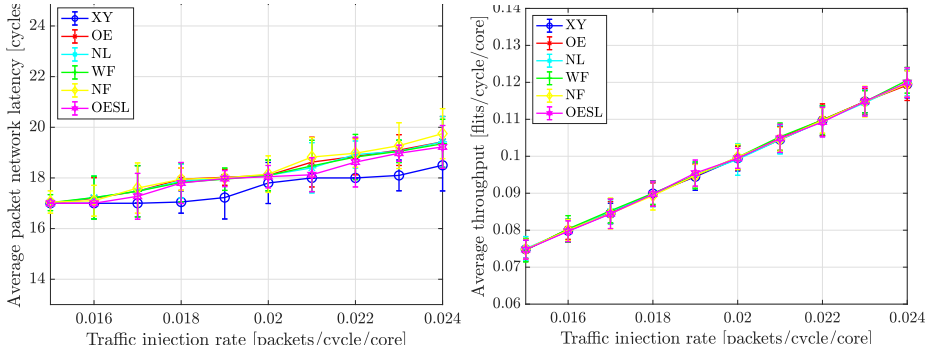
It has been noted that XY routing performs better in Uniform traffic because it incorporates global long term information about the traffic pattern. However, the other algorithms select the routing paths based on local, short-term information. This decision benefits only the packets in the nearest future, which tend to interfere with other packets. Hence, the smoothness of Uniform traffic is not necessarily maintained in the long term. However, for most of the real world applications, each node will communicate with some nodes more frequently compared to others [Kundu and Chattopadhyay, 2018]. The XY routing is unable to deal with such non-uniform traffic patterns because of its determinism. Precisely, XY routing maintains the irregularity of the non-uniform traffic, as it maintains the smoothness for the Uniform traffic [Hu and Marculescu, 2004]. This is obvious from the following figures under Transpose and BitReverse traffic.

As it is previously mentioned, in the BitReverse traffic scenario, the difference between latency and throughput measurements is less significant in most of 4×4 and 2×2 topologies (Figure 4.17, Figure 4.18). However, in the scenario of 8×8 topology the average packet latency (Figure 4.19(a)) it was between $20 - 80$ cycles and the average throughput between $0.06 - 0.13$ (Figure 4.19(b)). Precisely, it is obvious from the graphs that OESL outperforms the other routing algorithms by achieving the lowest latency and the highest throughput. Particularly, under the highest injection rate OESL achieved 17% better latency and 19% better throughput than the classic OE.



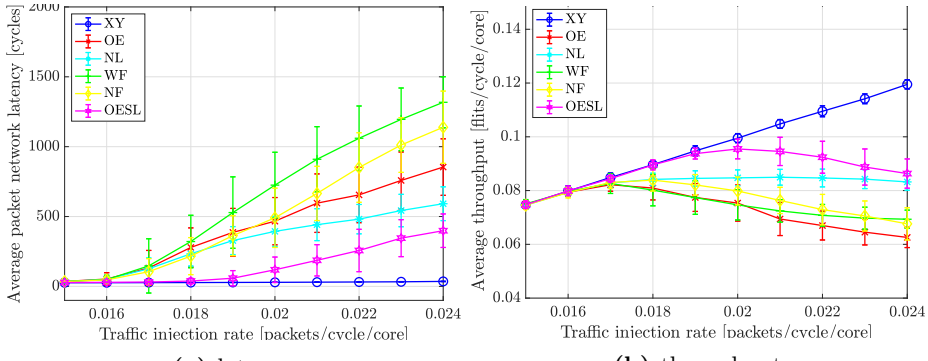
(a) latency **(b)** throughput

Figure 4.14: 95% coverage of mean values under Uniform traffic (Topology: 2x2 Mesh).



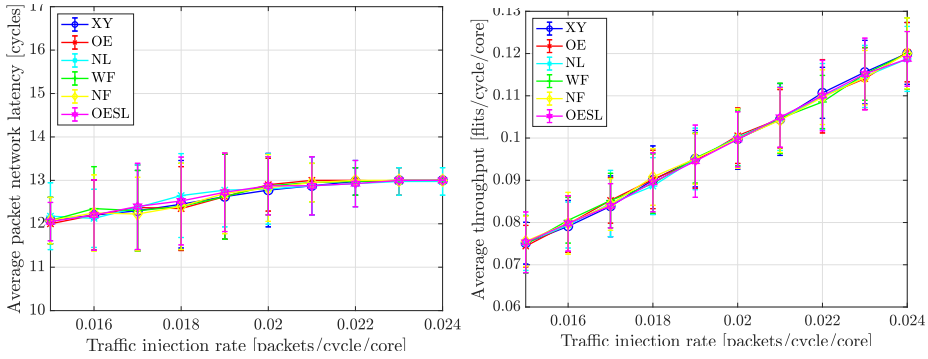
(a) latency **(b)** throughput

Figure 4.15: 95% coverage of mean values under Uniform traffic (Topology: 4x4 Mesh).



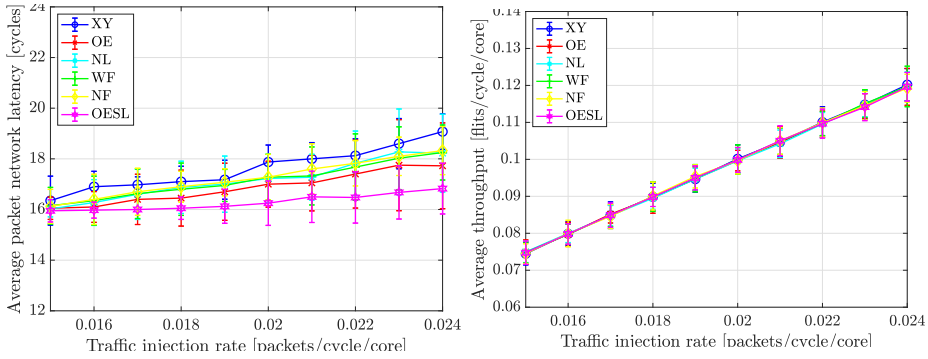
(a) latency **(b)** throughput

Figure 4.16: 95% coverage of mean values under Uniform traffic (Topology: 8x8 Mesh).



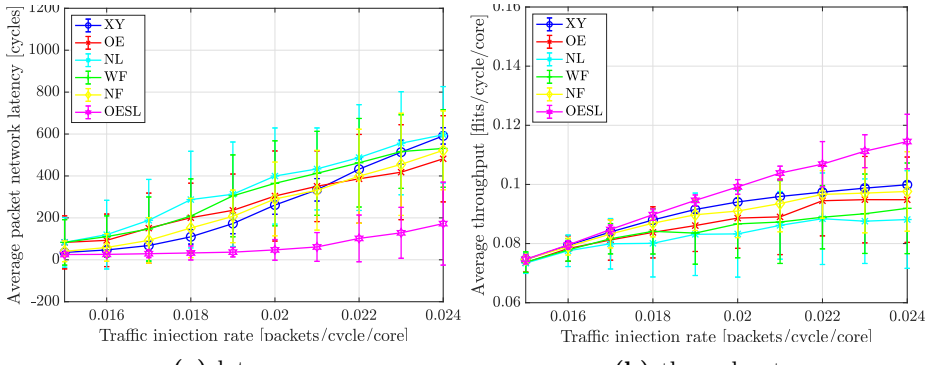
(a) latency **(b)** throughput

Figure 4.17: 95% coverage of mean values under Bit Reverse traffic (Topology: 2x2 Mesh).



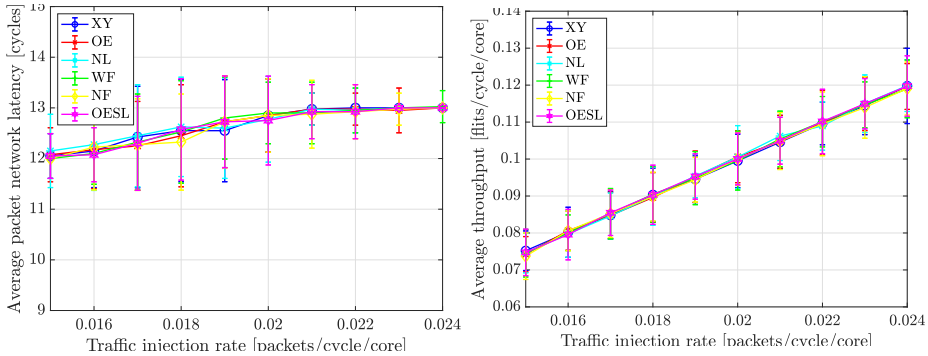
(a) latency **(b)** throughput

Figure 4.18: 95% coverage of mean values under Bit Reverse traffic (Topology: 4x4 Mesh).



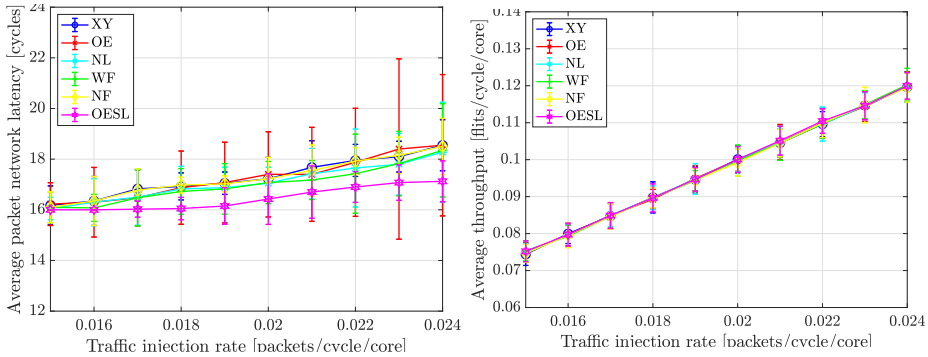
(a) latency **(b)** throughput

Figure 4.19: 95% coverage of mean values under Bit Reverse traffic (Topology: 8x8 Mesh).



(a) latency **(b)** throughput

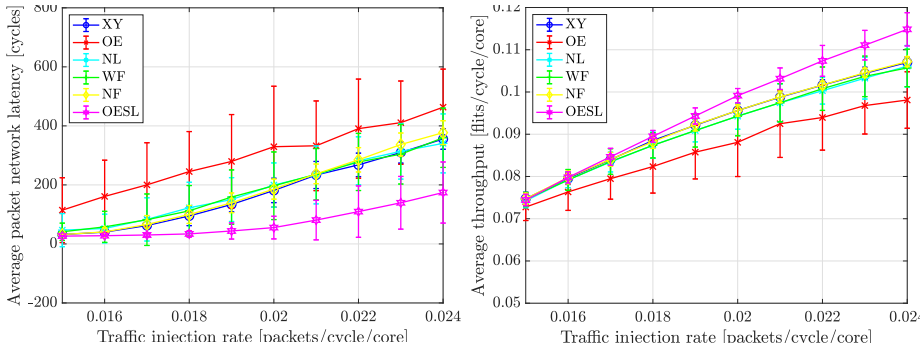
Figure 4.20: 95% coverage of mean values under Transpose traffic (Topology: 2x2 Mesh).



(a) latency **(b)** throughput

Figure 4.21: 95% coverage of mean values under Transpose traffic (Topology: 4x4 Mesh).

Similarly, in the Transpose traffic scenario the difference between latency and throughput measurements is less significant in most of 4×4 and 2×2 topologies (Figure 4.20, Figure 4.21). Nevertheless in an 8×8 topology the average packet latency (Figure 4.22(a)) was between $20 - 60$ cycles and the average throughput between $0.06 - 0.12$ flits/cycle/core (Figure 4.22(b)). Precisely, under the highest injection rate OESL achieved 10% better latency and 16% better throughput than the classic OE.



(a) latency **(b)** throughput

Figure 4.22: 95% coverage of mean values under Transpose traffic (Topology: 8x8 Mesh).

4.5 Analysis of variances

There are a lot of inputs and outputs in the implemented scenarios, therefore the impact of each input on the perfomance of the network is a very interesting research topic to investigate. Hence, the ANOVA technique is selected.

4.5.1 Background

ANOVA is a statistical technique which is used to check if the means of two or more groups are significantly different from each other. It was developed by statistician and evolutionary biologist Ronald Fisher [Scheffe, 1999]. In its simplest form, ANOVA provides a statistical test of whether two or more population means are equal, and therefore generalizes the t-test beyond two means.

A system is referred to as multivariate when there are multiple dependent input variables. The input variables are referred to as factors. A factor x can be any variable (or parameter) which has probably an influence on the studied phenomenon. The factors are considered as a possible cause of the of the system behavior. The discrete values taken by the factors are called levels.

The response of the system is the set of output values that can be measured or applied to the studied phenomenon. The response is a direct consequence of the level of the factors that are injected as an input of the experiment. For example, if a system with a discrete-value factor a is considered to conduct k different experiments. On this system i represents the

observation number of experiments, and j represents a different level of the predictor variable x_{ij} with ($1 \leq j \leq n$ and $1 \leq i \leq k$). The matrix x_{ij} is called the matrix of experiments. Furthermore, the chance of an error during the experiments represented as ϵ. If an assumption that the system model is linear is made, the response obtained during i-th experiment can be mathematically expressed as:

$$y_i = a_0 + \sum_{j=1}^{\mathrm{n}} a_j \, x_{ij} + \sum_{j,k\neq 1}^{n} a_{jk} x_{ij} x_{ik} + \epsilon_i \tag{4.3}$$

or, as a linear system of equations:

$$\begin{aligned} Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} &= \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{11}x_{12} & x_{11}x_{13} & \dots \\ 1 & x_{21} & x_{22} & \dots & x_{21}x_{22} & x_{21}x_{23} & \dots \\ 1 & x_{31} & x_{32} & \dots & x_{31}x_{32} & x_{31}x_{33} & \dots \\ 1 & x_{41} & x_{42} & \dots & x_{41}x_{42} & x_{41}x_{43} & \dots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{n1}x_{n2} & x_{n1}x_{n3} & \dots \end{bmatrix} + \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \vdots \\ \epsilon_n \end{bmatrix} \\ &= a\Omega + \epsilon \end{aligned} \tag{4.4}$$

, where Y is the vector of experimental responses, Ω is the matrix of the model and is filled with the coefficients of the model, and ϵ is a vector that contains the experimental errors.

By using multivariate tests, such as ANOVA, it is possible to get information about the strength of the relationship between the factors and the corresponding responses. Two factors interact significantly if the performance response due to factor i depends on the value of the level j taken by the factor i. In other words, the relative change in the response can be observed if the second factor is modified.

The analysis of variance is based on the following assumptions:

(i) each population studied has the same variance,

(ii) the output scores for each input condition have to be normally distributed,

(iii) the observations have to be independent.

The key idea of ANOVA is to test the null hypothesis (H_0). The null hypothesis makes the assumption that the level of the output does not vary with respect to the input conditions, i.e. all experiments conducted with different levels of the input variables will have the same mean value of the output. If this hypothesis would have to be rejected, then it is possible to prove that, in reality, the input variable had a significant impact on the output. In order to express a formal mathematical definition of a value the interaction, the following definitions are introduced. First, the sum of squares for all the values taken together is equal to:

$$SS \triangleq \sum_{i=1}^{n}\sum_{j=1}^{k}(x_{ij}-\mu)^2 \tag{4.5}$$

, where μ is referred to as the grand mean of the samples. It can be written as:

$$\mu=\frac{1}{kn}\sum_{i=1}^{n}\sum_{j=1}^{k}x_{ij} \tag{4.6}$$

Due to the hypothesis that all of the values of the variance are nearly equal, it can be written:

$$\sigma_1^2 \simeq s_1^2, \sigma_2^2 \simeq s_2^2, \sigma_3^2 \simeq s_3^2, \dots \tag{4.7}$$

and the equation (4.7) can be rewritten as:

$$SS=\sum_{i=1}^{n}\sum_{j=1}^{k}(x_{ij}-\mu)^2=\sum_{i=1}^{n}\sum_{j=1}^{k}(x_{ij}-\mu_j)^2+n\sum_{j=1}^{k}(\mu_j-\mu) \tag{4.8}$$

, where μ_j is defined as the mean of the $j-th$ treatment. In the equation 4.8, the left part of the equation is called the Sum of Squares total (SS_{total}). The second sum of squares is referred to as the Sum of Squares within (SS_{within}) as it reflects the variation that occurs within each group. Furthermore the third sum of squares is called Sum of Squares between ($SS_{between}$) since it is based on the variation that occurs between groups. Therefore:

$$SS_{total}=SS_{within}+SS_{between} \tag{4.9}$$

Based on the Formula 4.7, the definitions of the mean squares can be introduced. The Mean of Squares (MS), is defined by dividing each of the three sums of squares by their respective degrees of freedom. It can be shown that, if the null hypothesis is true, the MS can be considered as it

estimates the variance of the considered population variance. On the other hand, if the null hypothesis is false, only the estimation of the population variance based on the value of SS_{within} would be a valid estimation since the other two sums would be modified due to effect of the treatment (i.e., the differences existing among the sample means). The total Mean of Squares is defined as:

$$MS_{error} \triangleq \sigma_e^2 = s_e^2 = \mathbb{E}(s_n^2) = \sum_n \frac{s_n^2}{k} \tag{4.10}$$

Now the treatment effect of a given input variable x_i can be defined as:

$$MS_A \triangleq \sigma_A^2 = \sum \frac{(\mu_i - \mu)^2}{k-1} \tag{4.11}$$

and the following expectations can be written, called test of the effect of Factor A (F) or $F - ratio$.

$$F \triangleq \frac{\mathbb{E}(MS_A)}{\mathbb{E}(MS_{error})} = \frac{\sigma_e^2 + n\sigma_A^2}{\sigma_e^2} \tag{4.12}$$

Under the null hypothesis H_0: $\mu_{A1} = \mu_{A2} = ... = \mu$. As a consequence, the value of variance of the effect of F will be $\sigma_A^2 = 0$. In this case, the $F - ratio$ will have an expected value of approximatively $F \simeq 1$ and will be distributed as the standard F distribution. However, if H_0 is false and, therefore, there is a significant interaction, the value of σ_A^2 will not be equal 0 and the value of $F - ratio$ will be as different from 1 as the level of interaction is high. More intuitively input, a variable (or a n-way interaction between n variables) is considered to be significant if the variation of its level has a significant impact on the output values of the system; which is denoted by the amplitude of its corresponding $F - ratio$.

The $F - statistics$ represents the level of significance of the interaction. When the $F - ratio$ is high, the interaction is significant. On the other hand, for lower values of F, it is unlikely for significant interaction to take place. However, particular data can exist where the null-hypothesis H_0 could be rejected, but showing only a small difference with the values. Such a case leads to a wrong rejection of the null hypothesis and to an incorrect interaction statement.

In statistics, two types of errors are defined. Firstly, the null hypothesis can be rejected, i.e., the idea that there is no interaction can be rejected, while in fact there is. This kind of error is called Type I error and its conditional probability (the probability of rejecting the null hypothesis given

that it is true) is designated as α, the size of the rejection region. Secondly, another error can be made by failing to reject the null hypothesis when it is in fact false. This type of error is called a Type II error, and its probability is symbolized by β. The power of the test is defined as the probability of rejecting $H0$ when it is actually false. Table 4.2 synthesizes the possible outcomes of the decision making process and the associated types of errors.

Table 4.2: Possible outcomes after decision process within ANOVA

Decision	$H_0 : True$	$H_0 : False$
$H_0 : Rejection$	$Type\ I\ Error$ $p = a$	$No\ Error$ $p = 1 - \beta \triangleq power$
$H_0 : No\ Rejection$	$No\ Error$ $p = 1 - a$	$Type\ II\ Error$ $p = \beta$

In Figure 4.23, two distribution are presented: $H0$ (blue curve) and $H1$ (white curve). The distribution $H0$ is defined as the sampling distribution of the mean value of the output when the null hypothesis is verified. On the other hand, the distribution $H1$ is the sampling distribution of the mean value of the output as it is observed (i.e. without any hypothesis about the interaction of the variables). It is important to note that the distributions $H0$ and $H1$ are drawn on the hypothesis that the distribution of the observed samples is normal. As a consequence, its mean value μ and its variance σ_2 can be computed and the values of α and β are obtained from table. Furthermore when the distance between μ_1 and μ_0 increases, the power of the test substantially increases too. Generally, a typical value for the power the test is $p < 5\%$.

4.5.2 Scenarios-Results

For the ANOVA the Matlab program is used. Firstly the input data were reformed and afterwards 1-way ANOVA was performed between latency and throughput and each factor separately: tir, routing, traffic. Secondly, a multi-comparison with a N-way ANOVA tests was performed in order to determine which pairs of groups of means are significant different.

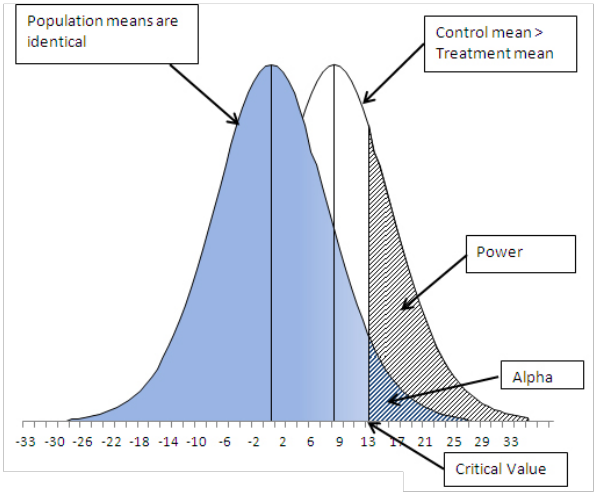


Figure 4.23: Graphical representation of the power of rejection of the null hypothesis.

4.5.3 One-way ANOVA

The function $p = anova1(y)$ returns the $p - value$ for a balanced one-way ANOVA. It also displays the standard ANOVA table and a box plot of the columns of y. The function of ANOVA tests the hypothesis that the samples in y are drawn from populations with the same mean against the alternative hypothesis that the population means are not all the same. The results of the one-way ANOVA are depicted on the Table 4.3. As a first conclusion, it can be seen that all one-way interactions are found to be significant for the two considered performance metrics (throughput, latency). More specifically, regarding the latency, it can be observed that the main effect comes from the choice of the traffic ($F = 26.87$, $p < 0.05$) by following the routing ($F = 6.6$, $p < 0.05$) and tir ($F = 5.52$ $p < 0.05$). Regarding the throughput, the main effect comes again from the choice of the traffic ($F = 12.86$, $p < 0.056$) by following the tir ($F = 9.87$, $p < 0.05$) and last but not least the routing ($F = 6.31$, $p < 0.05$). Furthermore, from the $p - value$ it is obvious that in every scenario the null hypothesis can be rejected since $p < 5\%$.

Figure 4.24 and Figure 4.25 depict the boxplot of the one-way ANOVA. On each box, the central mark is the median and the edges of the box are the $25th$ and $75th$ percentiles (1st and 3rd quantiles). The whiskers extend to the most extreme data points that are not considered outliers. The outliers are plotted individually. The interval endpoints are the extremes of

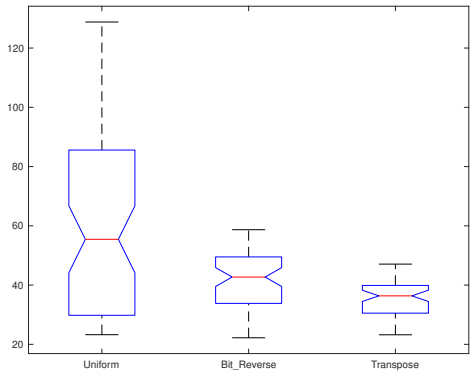
the notches. The extremes correspond to $q2 - 1.57(q3 - q1)/sqrt(n)$ and $q2 + 1.57(q3 - q1)/sqrt(n)$, where $q2$ is the median (50th percentile), $q1$ and $q3$ are the $25th$ and 75th percentiles, respectively, and n is the number of observations without any NaN values. Two medians are significantly different at the 5% significance level if their intervals do not overlap. This test is different from the $F - test$ that ANOVA performs, but large differences in the center lines of the boxes correspond to large $F - statistic$ values and correspondingly small $p - values$.

Table 4.3: One way ANOVA

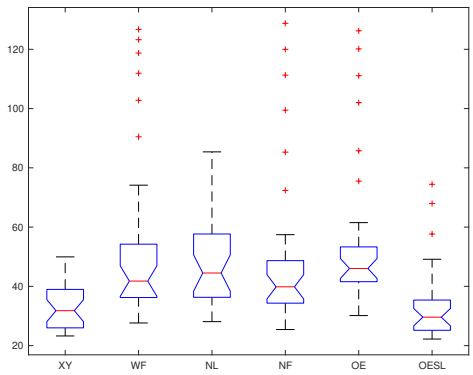
Source of Interaction	**Latency**		**Throughput**	
	F-ratio	**p-value**	**F-ratio**	**p-value**
tir	5.52	1.17632e-06	9.87	4.17398e-12
routing	6.6	1.1839e-05	6.31	2.0826e-05
$traffic$	26.87	6.44187e-11	12.86	6.10229e-06

By performing a multiple comparison of the mean latency of routing, in the Figure 4.26(b), the blue bar represents the comparison interval for mean latency for XY. The red bars represent the comparison intervals for the mean latency for WF, NF and OE. Neither of the red bars overlap with the blue bar, which indicates that the mean latency of XY is significantly different from that of WF, NF and OE. By clicking on the other routing algorithms the results that has been obtained are the following: a)the mean latency of WF is significantly different from that of XY and OESL, b)the mean latency of NL is not significantly different from the rest of routing algorithms, c) the mean latency of NF is significantly different from that of XY and OESL, d)the mean latency of OE is significantly different from that of XY and OESL, e)the mean latency of OESL is significantly different from that of WF, NF and OE.

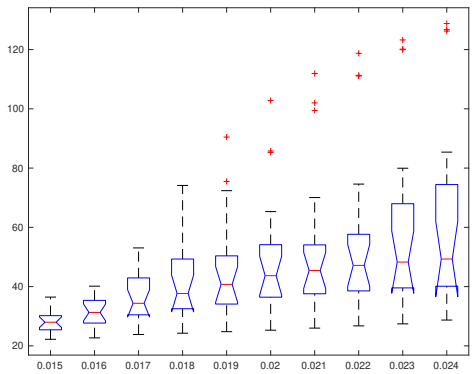
In the same way it is performed multi-comparison of the mean latency of tir and traffic, the results are depicted on Figure 4.26. As far as the traffic is concerned: a)the mean latency of Uniform traffic is significantly different from that one of the Transpose and BitReverse, b)the mean latency of BitReverse traffic is significantly different from that one of Uniform traffic. c)the mean latency of Transpose traffic is significantly different from that one of Uniform traffic. As far as the tir certain values of it present significant deference between other values, like the mean latency of tir=0.015 is significantly different of this one of 0.021, 0.022, 0.023, 0.024, and certain values of tir presented no significant difference with others, like the mean latency of tir 0.018 has no significantly difference from other tir.



(a) Latency-Traffic

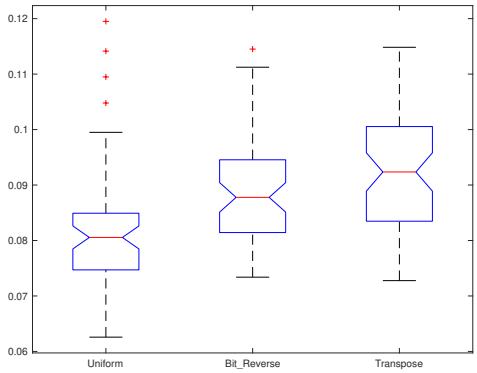


(b) Latency-Routing

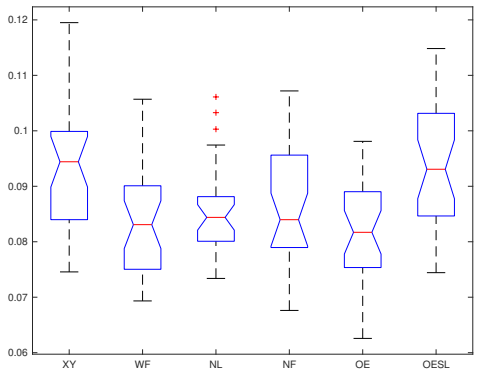


(c) Latency-Tir

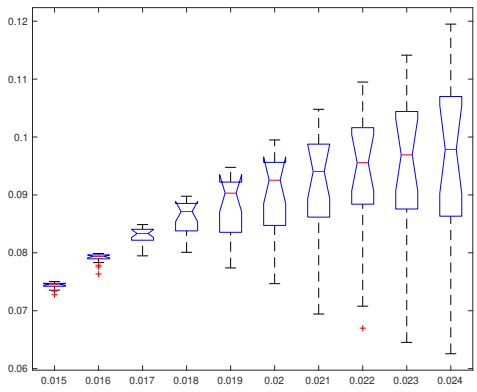
Figure 4.24: One-way Anova boxplot



(a) Throughput-Traffic

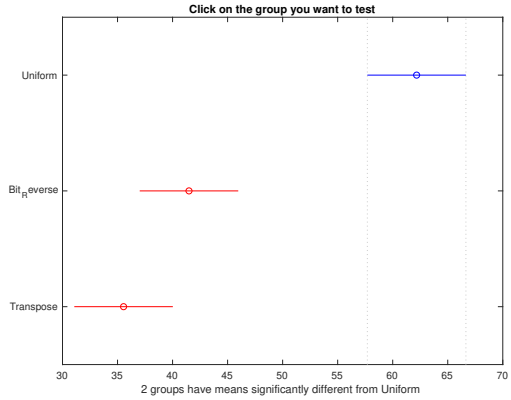


(b) Throughput-Routing

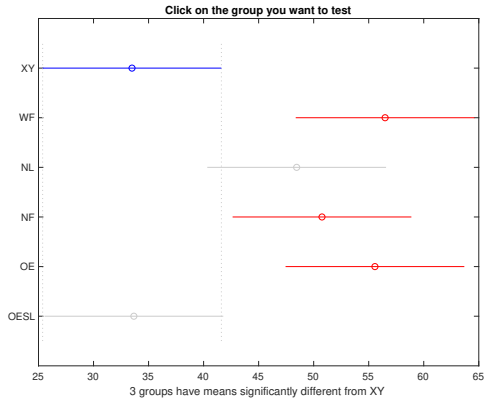


(c) Throughput-Tir

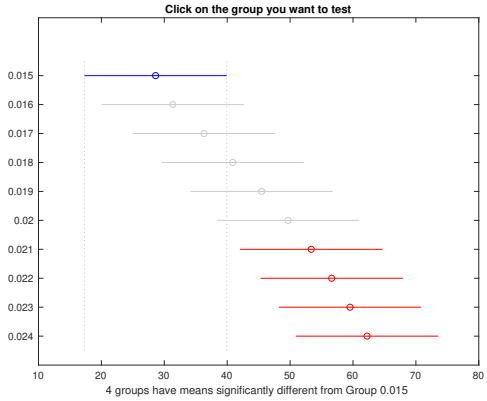
Figure 4.25: One-way Anova boxplot



(a) Latency-Traffic

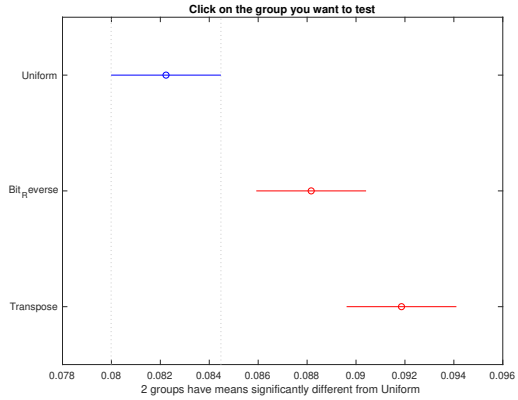


(b) Latency-Routing

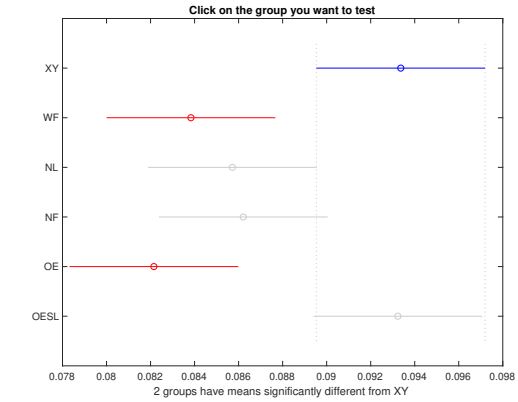


(c) Latency-Tir

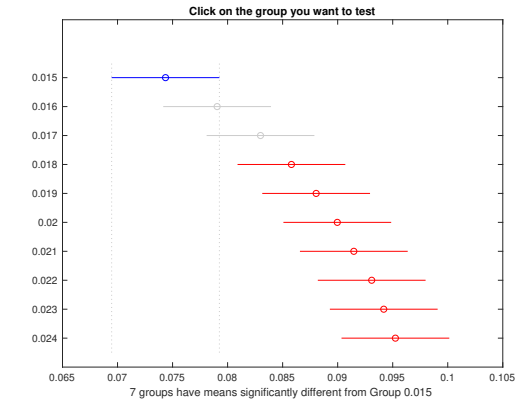
Figure 4.26: Multiple comparison of the mean latency of routing, traffic, tir



(a) Throughput-Traffic



(b) Throughput-Routing



(c) Throughput-Tir

Figure 4.27: Multiple comparison of the mean throughput of routing, traffic, tir

Furthermore, the same tests have been performed also for the mean throughput and the results are depicted on Figure 4.27. The results were similar with some minor changes. As far as the traffic results of multi-comparison test, the results were exactly the same. Concerning the routing, the following difference are observed : a)the mean latency of XY is significantly different from that of WF and OE, b)the mean latency of NF is not significantly different from the rest of routing algorithms, c)the mean latency of OESL is significantly different from that of WF and OE. Regarding the tir, the mean latency of most of values were significantly different from the others.

4.5.3.1 N-way ANOVA

The $p = anovan(y, group)$ returns a vector of $p - values$, one per term, for multiway (n-way) ANOVA for testing the effects of multiple factors on the mean of the vector y. N-way ANOVA is a generalization of two-way ANOVA. For three factors, for example, the model can be written as:

$$y_{ijkr} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \epsilon_{ijkr} \quad (4.13)$$

, where y_{ijkr} is an observation of the response variable. i represents group i of factor A, $i = 1, 2, ..., I$, j represents group j of factor B, $j = 1, 2, ..., J$, k represents group k of factor C, and r represents the replication number, $r = 1, 2, ..., R$. For constant R, there are a total of $N = I * J * K * R$ observations, but the number of observations does not have to be the same for each combination of groups of factors. μ is the overall mean. α_i are the deviations of groups of factor A from the overall mean μ due to factor A. The values of α_i sum to 0. β_j are the deviations of groups of factor B from the overall mean μ due to factor B. The values of β_j sum to 0.
γ_k are the deviations of groups of factor C from the overall mean μ due to factor C. The values of γ_k sum to 0. $(\alpha\beta)_{ij}$ is the interaction term between factors A and B. $(\alpha\beta)_{ij}$ sum to 0 over either index. $(\alpha\gamma)_{ik}$ is the interaction term between factors A and C. The values of $(\alpha\gamma)_{ik}$ sum to 0 over either index. $(\beta\gamma)_{jk}$ is the interaction term between factors B and C. The values of $(\beta\gamma)_{jk}$ sum to 0 over either index. $(\alpha\beta\gamma)_{ijk}$ is the three-way interaction term between factors A, B, and C. The values of $(\alpha\beta\gamma)_{ijk}$ sum to 0 over any index. ϵ_{ijkr} are the random disturbances. They are assumed to be independent, normally distributed, and have constant variance.

The results of N-way ANOVA are shown in Table 4.4. As far as the the latency, the main effect comes from the choice of the traffic ($F = 120.95$, $p = 0$), which is obvious from the results the One-way ANOVA. As far as the

interaction of the factors, the pair $traffic * routing$ has the biggest effect on latency ($F = 22.82$, $p = 0$). The $p - values$ for interaction $tir * routing$ is much larger than a typical cutoff value of 0.05, indicating these terms are not significant. As far as the throughput, the main effect comes from the choice of the traffic ($F = 68.61$, $p = 0$) which is obvious from the results of the one-way ANOVA. As far as the interaction of the factors, the pair $traffic * routing$ has the biggest effect on throughput ($F = 10.79$, $p = 0$), which is obvious from the latency results. However the $p - values$ for interaction $tir * routing$ it is lower than a typical cutoff value of 0.05, indicating these terms are significant in contrast to the latency results.

Table 4.4: Results of N-way ANOVA for latency and throughput

Source of Interaction	**Latency**	**Throughput**	**Latency**	**Throughput**	**Latency**	**Throughput**	**Latency**	**Throughput**
	SS		**MS**		**F-ratio**		**p-value**	
tir	22863	0.00766	2540.3	0.00085	35.24	41.24	0	0
traffic	23559.4	0.00283	11779.7	0.00142	163.39	68.61	0	0
routing	161131.3	0.00342	3226.3	0.00068	44.75	33.2	0	0
$tir * traffic$	12437.2	0.00268	691	0.00015	9.58	7.21	0	0
$tir * routing$	3224.6	0.00164	71.7	0.00004	0.99	1.76	0.4979	0.0114
$traffic * routing$	16449.1	0.00223	1644.9	0.00022	22.82	10.79	0	0
error	6488.7	0.00186	72.1	0.00002				

4.6 Summary-Discussion

In this chapter, firstly some of the most used NoC simulators, which are introduced in literature were discussed in detail by highlight their characteristics. By defining some high importance parameters for the proposed SDNoC architecture, an overview of the NoC simulators was presented. The Garnet2.0 simulator was chosen due to its reconfigurability and flexibility on routing, topologies and traffic but also because it provides full system simulation with more realistic results. Thereafter, the implementation of the SDNoC prototype and the MicroLET communication protocol was demonstrated. Furthermore, the changes that has been made, within Garnet and Gem5 simulators, in order to implement the SDNoC prototype were explained in detail. A performance evaluation of different implemented routing algorithms (XY, OE, NL, WF, NF, OESL), within SDNoC architecture, under different scenarios was shown. The scenarios are consisted of 3 topologies, under 3 different traffic models and under multiple injection rates.

Routing problems are diverse by having negative impact on network performance. The main challenge behind interesting on NoC or SDNoC routing is to increase the reliability of the network, while ensuring a sensible performance. The main evaluated performances are: low latency, high throughput and low power consumption. However, many papers present experiments evaluating different routing algorithms, when it comes for NoC the routing algorithms are architecture oriented. With the aforementioned results, an effort has been made in order to show how the different routing algorithms were performing within SDNoC by providing realistic results under different routing scenarios. One research question that raised was how the performance is affected by the different parameters, for this reason it is performed for first time in the context of NoC an analysis of variance in order to show the interaction of the different parameters within the network. Precisely, the ANOVA of the SDNoC different factors: tir, traffic, routing and their interactions was presented.

ANOVA is a statistical technique that is used to check if the means of two or more groups are significantly different from each other. Furthermore ANOVA checks the impact of one or more factors by comparing the means of different samples. Briefly, the conclusions that were drawn through ANOVA are: a) the one-way interactions of the factor: $traffic$ is found to be the most significant for the two considered performance metrics (latency, throughput), following the $routing$ and tir, b) the n-way interactions the pair of the factors: $traffic * routing$ have significant affect on the

latency and throughput.

As far as the routing results is concerned, under Uniform traffic, the proposed routing algorithm but also the rest of the routing algorithms have lower performance than XY routing. This is due to unreliability of the Network Monitoring Phase (Section 3.6.3). On the other hand, under the Transpose and BitReverse traffic, the proposed routing algorithm outperforms the rest of routing algorithms. Indeed, under such traffic scenarios, the controller relies on an accurate view of the network state and it is able to balance the traffic across the network by avoiding the form of congested network areas. Conversely, under these scenarios, XY pushes the traffic towards the same links and switches. Therefore, the corresponding network areas become congested, which leads to a network performance decrease.

The proposed routing algorithm OESL relies on the OE algorithm, which is partially adaptive and therefore, restricts the number of admissible routes. Secondly, the controller responds to the arriving `ROUTE_REQUEST` messages by allocating routes without being aware of future routes. Thereby, when the controller searches to allocate a route for a source-destination pair, it is possible that all the admissible routes being occupied, while there are still possible routes (but not admissible) flowing through unoccupied resources. This scenario can happen during some simulations according to the traffic. If the controller will be aware beforehand about the future `ROUTE_REQUEST`, it could adapt the allocation of the routes accordingly in order to avoid this scenario. Moreover, the lack of knowledge concerning the future requests is less critical if the controller uses a fully adaptive routing. Despite the higher standard deviations, the proposed routing approach still outperforms the rest of the routing algorithms under Transpose and BitReverse traffic and it could be a possible solution for future SoCs.

Chapter 5

Security within SDNoC

5.1 Introduction

Security within Software Defined Network (SDN) is a very sensitive topic. As previously mentioned, one of the most used SDN-based communication protocols seen in literature is the OpenFlow Protocol [McKeown et al., 2008]. This protocol has several security flaws that can be exploited to compromise the network. Additionally, the SDN paradigm is susceptible to several security breaches [Zhang et al., 2018a]. The existing security solution of Transport Layer Security (TLS) protocol is not well enforced in the current version of the OpenFlow standard [Foundation, 2015]. The lack of TLS use could lead to fraudulent data insertion and Denial of Service (DoS) attacks [Benton et al., 2013]. Under the umbrella of Public Key Cryptography (PKC), the TLS protocol requires a Certification Authority (CA) to generate the CA's key, certificates for the controllers, routers, and then the signing of these certificates with the CA's key. Afterwards, the certificates and the keys of the network entities are deployed to the respective devices. However, in this research, the main concern is the communication security among Software Defined Network-on-Chip (SDNoC) routers and the controller, which is an unexplored topic. As far as the Public Key Infrastracture (PKI) overhead, it includes generation and signing of digital certificates for the routers and the controller. This makes PKI based solutions less attractive for and Multi Processor System-on-Chip (MPSoC) architecture. Therefore, a more suitable solution that fits the unique characteristics of the MPSoC architecture is needed. Precisely, in this chapter the design of new SDN-based protocol is discussed. This protocol has three main functionalities: the derivation of keys for every node in the network through a Private Key Generator (PKG), the establishment of a secure group of participants, and the secure communication between the partici-

pants is presented.

Since the number of processors and cores on a single chip is increasing, the interconnection among them becomes significant. As shown earlier, Network-on-Chip (NoC) has direct access to all resources and information within a System-on-Chip (SoC), rendering it appealing to attackers. Malicious attacks targeting NoC are a major cause of performance depletion and they can cause arbitrary behavior of links or routers, that is, Byzantine faults. Byzantine faults have been thoroughly investigated in the context of Distributed systems, however not in Very Large Scale Integration (VLSI) systems. Hence, in this chapter an introduction on to Byzantine faults, together with a novel fault model is presented, followed by the design of 2 lightweight algorithms for tolerating the Byzantine faults, based on SDNoC architecture.

Following the fault model of the Byzantine faults within the SDNoC, malicious attacks and malicious hardware modifications of a circuit during the design or fabrication often lead to arbitrary failures and can cause faulty nodes to exhibit arbitrary behavior, these are Byzantine failures. Byzantines failures occur when the system is under specific attacks like Hardware Trojan (HT), DoS, HT-DoS, etc. In this thesis a novel HT-DoS attack, called Greyhole attack is introduced. The HT-Greyhole attack targets the routers within NoC by forcing them to block certain packets instead of forwarding them. This, lead to performance decrease and packet loss increase. Furthermore, during a HT-Greyhole attack, only certain packets that are arriving at the router, are dropped making it hard to detect. Furthermore, a detection and defense method against HT-Greyhole attack, based on the SDNoC architecture, is presented.

Firstly, by following the SDN concept, a new security protocol, which is called Secure Sdn-based Protocol over mpSoC (SSPSoC), is proposed in order to secure the communication and efficiently manage the routing within the MPSoC. The SSPSoC includes a private key derivation phase, a Group Key Agreement (GKA) phase, and a data exchange phase in order to ensure that basic security primitives are preserved and provide secure communication. Afterwards, an introduction to Byzantine faults within SDNoC is discussed by following a new fault model. Also, a novel algorithm relying on SDNoC for tolerating the Byzantine faults is explained. Lastly,

the description and activation of an a novel HT-Greyhole attack in NoC context together with a detection and defense method is introduced.

5.2 Secure Sdn-based Protocol over mpSoC

The existing literature on SDN security refers to point-to-point communication between routers and a controller. However, running different applications on a MPSoC creates multiple routing paths among Processing Elements (PEs) and therefore multiple routers interacting with the controller quite frequently. Generally, the application based logical subsystems will be created, which may involve multiple ICs (with different components such as GPU, crypto processor etc.). The main idea is to create a secure point-to-multipoint communication between a controller and a group of routers, i.e. secure multicast. Therefore the SSPSoC [1] is proposed, which includes three phases: 1. Obtain Private Key, 2. From a Group and 3. Router controller communication. As fas as the second phase, two GKA protocols are chosen to be tested within the proposed SDNoC architecture: [Sharma et al., 2017] and [Teng and Wu, 2016]. The protocols that were chosen are balanced and imbalanced GKA protocols respectively. In balanced GKA protocols, all the participating nodes share the same computational burden while in imbalanced protocols, the powerful node (in this case, the controller) is mainly responsible for expensive computations. Sharma's protocol comes with the benefit of achieving mutual authentication using signature and of course, the assurance that every participant at the end is in possession of the valid group key. However, the Teng protocol does not provide mutual authentication, thus trading security for efficiency.

5.2.1 Security Requirements

The SDNoC architecture contains multiple routers and a single controller to manage the overall communication among PEs. The infrastructure to implement identity based cryptography requires an on-board PKG. The overall communication security on this layer (router-controller) can be investigated from two viewpoints. The first view is to securely deliver the private keys to the routers and the controller. The second view covers the secure communication between all the routers and the controller. In or-

[1] This contribution has been done under collaboration of Dr. Gaurav Sharma and Ir. Theofanis Rigas. More precisely the part of Security Requirements has been done by Dr. Sharma and the Group Key Agreement part has been done by Dr. Gaurav Sharma and Ir. Thefanis Rigas. The rest of the contribution is a collaboration between Ir. Soultana Ellinidou, Ir. Theofanis Rigas and Dr. Sharma.

der to achieve this security, an authenticated group key agreement protocol should be used.

5.2.1.1 Phase 1

The foremost issue to address is to transport the Private Key (PK) and the required security parameters to all routers and the controller. The PKG generates a PK for all routers and the controller and delivers it securely. The literature suggests using a secure channel but they do not specify exactly what this channel could be and its security requirements. The possible threats and solutions are:

- In order to ensure that the only legitimate nodes can receive identity ID and PK, node authentication must be performed by the PKG.
- A counterfeit PKG with a different master key can generate private keys and IDs for the nodes. This PKG is able to decrypt all the traffic between nodes and controller. In this case, *authentication* of the PKG by the nodes is also needed.
- An attacker can eavesdrop the response of the PKG and steal the PK of a node. A solution must be there to ensure the *confidentiality* of communication between a node and the PKG.
- An attacker can *sniff* the packets exchanged between a node and the PKG and replay them later to obtain a PK.
- An attacker can compromise the *integrity* of the packets between node and PKG.

5.2.1.2 Phase 2

This phase refers to router-controller group communication where a GKA protocol is adopted. The common threats are spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privileges. The authenticated GKA protocol provides authentication of all participants. As the session key is derived, rest of the communication is encrypted using AES-GCM, which is widely adopted within NoC [Cotret et al., 2016], with session key. Therefore, confidentiality, authenticity, integrity and non-repudiation are ensured. To address denial of service and authorization issues, separate precautions need to be enabled.

5.2.2 Group Key Agreement

5.2.2.1 Assumptions

Before the design of the protocol some vital assumptions are required :

- The network consists of multiple nodes, which can be either a controller or routers or the PKG, which derives the private keys to the network entities.
- The private ID-based keys are provided to the participants of the group by PKG, which is the private key generator. Supposing that msk is the master secret key of the PKG and $KPub_{PKG}$ is the public key.
- Given that there are n entities in the network, $U_1, U_2, ..., U_n$. These entities form a group of participants in a GKA session for establishing a group key. Assuming that U_n will be the controller of every group, there are $n-1$ nodes (routers) in this group.

5.2.2.2 Group Key Agreement Protocols

By taking into account the above assumptions and the security requirements, the two selected GKA protocols are presented. These two protocols consist of 3 major phases: The *Setup* phase (hash functions, group generators, pairing), the *Key Extraction* phase (nodes and controller obtain their private keys from PKG) and the *Key Agreement* phase (establishment of a group session key for participants).

The Teng protcol [Teng and Wu, 2016]:

Setup:

- The PKG chooses two groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of prime order q, a bilinear pairing [Boneh and Franklin, 2001] $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$.
- The PKG selects two random generators P and Q of $\mathbb{G}_1$.
- ThePKG selects $s \in \mathbb{Z}_q^*$ as the msk and sets $KPub_{PKG} = sP$.

Private Key Extraction: Defining as input parameters, msk and $ID_i \in \{0,1\}^*$ with ID_i being the ID of the node:

- The PKG computes $KPriv_i$ as $S_i = (q_i + s)^{-1}Q$ where $q_i = H(ID_i)$.
- The PKG communicates secretly $KPriv_i$ to node i.

- The public key of node i is $T_i = q_iP + KPub_{PKG} = (q_i + s)P$.

Key Agreement Round 1: Each participant:

- U_i selects a random $r_i \in \mathbb{Z}_q^*$.
- U_i pre-computes $P_i = r_iT_i$.
- U_i $(1 \leq i \leq n)$ sends P_i to the controller S.

Key Agreement Round 2: Upon receiving P_i from all nodes, each participant:

- U_i $(1 \leq i \leq n)$, S chooses random $r \in \mathbb{Z}_q^*$.
- U_i computes $Q_i = rP_i$.
- U_i broadcasts Q_i $(1 \leq i \leq n)$, keeping r secret.

Key Computation: On receiving Q_j $(1 \leq j \leq n)$:

- U_i computes the final session key as
  $sk = e(Q_i, S_i)^{r_i^{-1}} e(Q_1 + Q_2 + ... + Q_n, Q)$
  $= e(P, Q)^{r+rr_1(s+q_1)+...+rr_n(s+q_n)}$.

Pre-Computation: The following tuples (r_i, r_i^{-1}, P_i) should be created and stored in the memory storage of the nodes before the execution of the GKA. This essentially reduces the computation cost of the first round for the nodes and also improves the speed in the key computation phase.

The Sharma protocol [Sharma et al., 2017]:

Assumption: Let *pid* be the set of the identities of the participants in one session of the protocol and *sid* the session identifier.

Setup:

- The PKG selects an EC group $\mathbb{G}$ of prime order q. Let P be a generator of group $\mathbb{G}$.
- The PKG computes the system's public key as $KPub_{PKG} = sP$ by choosing a master secret $s \in \mathbb{Z}_q^*$.
- The PKG chooses cryptographic hash functions $H_1 : \{0,1\}^* \times \mathbb{G} \rightarrow \mathbb{Z}_q^*, H_2 : \{0,1\} \times \mathbb{G} \rightarrow \mathbb{Z}_q^*$ and $H_3 : \{0,1\}^* \rightarrow \{0,1\}^k$.

Key Extraction: Defining the system parameters $Params = \{G, q, H_1, H_2, H_3, H, KPub_{PKG}\}$ and by keeping the master key secret:

- The PKG selects $r_i \xleftarrow{\$} \mathbb{Z}_q^*$ and computes $R_i = r_i P$.
- ThePKG computes the private key for the user U_i as $KPriv_i = r_i + sH_1(ID_i, R_i)$.
- Each participant U_i can verify the private key as $KPriv_i P = R_i + H_1(ID_i, R_i) KPub_{PKG}$.

Key Agreement Round 1: Each participant:

- $U_i (1 \leq i \leq n)$ chooses $eph_i \xleftarrow{\$} \mathbb{Z}_q^*$ and computes $l_i = H_3(eph_i, KPriv_i)$ and $L_i = l_i P$.
- $U_i (1 \leq i \leq n)$ selects a random string $k_i \in \{0,1\}^k$. Each user, except U_n computes $H(k_i)$. The user U_n masks the randomness as $\tilde{k}_n = H(k_n, x_n)$ where x_n is the long-term secret of U_n.
- $U_i (1 \leq i \leq n)$ computes $H(\tilde{k}_n)$.
- $U_i (1 \leq i \leq n)$ broadcasts the tuple $< L_i, H(k_i), H(\tilde{k}_n), R_i >$ to all $n-1$ members.

Key Agreement Round 2: Upon receiving the message $< L_j, H(k_j), H(\tilde{k}_n), R_j >$, each participant:

- U_i computes $U_{ij} = l_i L_j$ and $L = L_1 || L_2 || .. || L_n$
- U_i, except U_n, computes $K_{ij} = H(U_{ij}) \oplus k_i$. the user U_n computes $mask = H(U_{ij}) \oplus \tilde{k}_n$
- Chooses another random number $t_i \in \mathbb{Z}_q^*$ and computes $T_i = t_i l_i P$. Also computes the signature on $< L, T_i >$ as $\sigma_i = t_i l_i + KPriv_i H_2(ID_i, L, T_i, pid)$
- Broadcasts $< K_{ij} (1 \leq j \leq n, j \neq i), mask, \sigma_i, T_i >$ to all $n-1$ members

Key Computation: Upon receiving $< K_{ji}, mask, \sigma_i, T_i >$, each participant:

- U_i verifies the received signature as: $\sigma_i P = T_i + (R_i + H_1(ID_i, R_i)$ $KPub_{PKG}) H_2(ID_i, L, T_i, pid)$
- U_i computes $\tilde{k}_j = H(U_{ji}) \oplus K_{ji}$. (Similarly, $\tilde{k}_n$ can be computed using mask.)
- Note that $U_{ij} = l_i L_j = l_i l_j P = l_j l_i P = l_j L_i = U_{ji}$.
- U_i checks the k_i as $H(k_j) = H(\tilde{k}_j)$ for $(1 \leq j \leq n, j \neq i)$.
- U_i computes the session identity $sid = H(k_i) || H(k_2) || ... || H(\tilde{k}_n)$.
- The session key is computed as $sk = H(k_1 || k_2 || ... || \tilde{k}_n || sid || pid)$.

5.2.3 Communication Protocol

In this section, the network architecture, followed by the packet format, the network messages that are broadcasted within the network, and the three phases of the proposed protocol are introduced.

5.2.3.1 Network Architecture

The SDNoC architecture consists of 3 main network entities:

- a PKG, which is considered as a trusted third party, generates the corresponding private key to the rest of the nodes(routers and controller).
- a centralized controller with a broader network view to manage the routing of packets within the network.
- multiple routers which are responsible to route the packets between the PE.

5.2.3.2 Packet Format

The packet format is the core of the protocol stack. Every packet consists of a header structure, which is 32-bits long, Figure 5.1, [Ellinidou et al., 2018]. The `header` message format consists of three main fields. Firstly, the `version` field indicates the version of communication protocol that is used for this message. Secondly, the `length` field indicates where this message will end in the byte stream starting from the first byte of the header. Thirdly, the `xid`, or transaction identifier, is a unique value used to match requests to responses. Furthermore, every message that travels across the network consists of the same header of 32-bits. However, the payload size depends on the length field that is provided through header message and it can vary according to the type of the message. Afterwards, the `Source_ID` and `Destination_ID` are included, which contain the source and destination number for the given packet. Another field is the `prio`, which indicates the priority of the packet . As far as the field `type`, a specific message stack is designed, which is presented below.

5.2.3.3 Network Messages

The different types of messages, which were designed and integrated into packet format depicted in Table 5.1. The SSPSoC protocol includes 8 types of messages with different content. These messages are flowing through the links between the network entities. In addition, the type value of the messages is used to distinguish the GKA protocol messages from other messages

that might be circulating on the network and one byte is used to encode the message type.

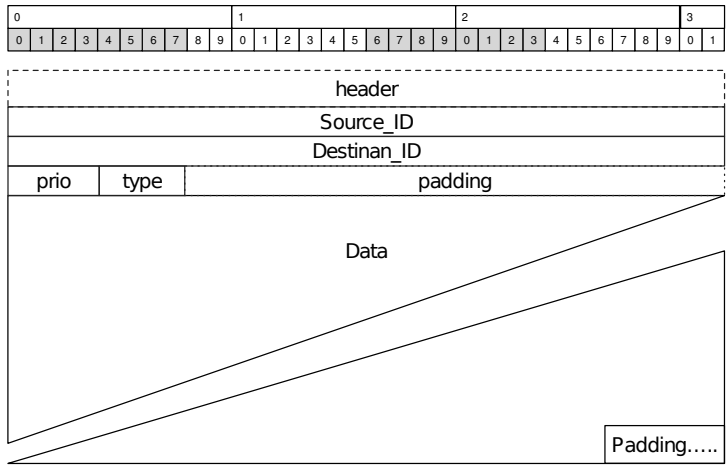


Figure 5.1: Packet format [Ellinidou et al., 2018]

Table 5.1: Designed Network messages

Type	Value	Description	Contents
`KEY_REQUEST`	0x06	Sent by nodes to the PKG	Enc(timestamp), IV, tag
`KEY_REPLY`	0x07	Sent by PKG to nodes as a reply to `KEY_REQUEST` message	Enc(System Parameters, node ID, private key), IV, tag
`JOIN`	0x01	Broadcasted by nodes who want to join a group	node ID, timestamp, JOIN token
`INVITE`	0x02	Broadcasted by controller for inviting nodes to form a group	(participant ID_1, node ID_1, ... , participant ID_n, $Node\ ID_n$, sid)
`READY`	0x03	Broadcasted by nodes as a reply to `INVITE` message	participant ID, timestamp, READY token, sid
`ROUND_1`	0x04	Contains cryptographic material for the first round of the GKA	sender ID, Crypto R1
`ROUND_2`	0x05	Contains cryptographic material for the second round of the GKA	sender ID, receiver ID, Crypto R2
`DATA`	0x08	Contains encrypted data with the group key	sender ID, receiver ID, Enc(data), IV, tag

5.2.3.4 SSPSoC Network Initialization

Phase 1: Obtain Private Key During the first phase, the routers and the controller communicate with the PKG, in order to obtain their long-term private keys. One of the major concerns on this phase is the security level of the communication between the nodes and the PKG. This problem could be solved by establishing a secure channel for the private key transmission. However, keeping the private key confidential is not the only security consideration, it should also be taken into account the authentication of the nodes. The authentication will ensure that only legitimate nodes can obtain a private key from the PKG. For this reason, the implementation of authenticated `KEY_REQUEST` messages is mainly used.

A node first determines a timestamp (t_s) to prevent the replay attacks [Syverson, 1994]. Afterwards it generates the random part of the Intialization Vector (IV) and it encrypts t_s using the Pre-Shared Key (PSK) and AES-GCM [Dworkin, 2007]. AES-GCM outputs the ciphertext c and the authentication tag: $c, tag = AES_{PSK,IV}(t_s)$. Thereafter the node sends a `KEY_REQUEST` message to the PKG, which contains the IV, the ciphertext and the authentication tag. It follows a process where the PKG decrypts the ciphertext and checks the authentication tag. If the tags are matching, it will check that the decrypted timestamp is within a given threshold. In case of the timestamp is valid, it will generate a random node ID and it will extract the associated private key, $KPriv(i)$. Thereupon, it generates a random IV and encrypts them using the PSK and AES-GCM and sends the IV, ciphertext, and the authentication tag to the node. The steps of the this procedure are depicted on Figure 5.2.

Phase 2: Form a Group On this phase, each router communicates with the controller in order to show interest in joining a group. The controller decides upon the group members and invites them to join the group by taking into account a session identifier (sid) for its session of the protocol.

Firstly, it is assumed that Phase 1 has already been performed and that the routers and controller have securely obtained their private keys. The controller has the power to decide which participants to invite to join a group, according to the network requirements and rules that are predefined by the user/designer. However when a router wants to join an already existing group, it broadcasts a `JOIN` message with its node ID, without knowing the ID of controller. The controller is waiting for `JOIN` messages in order to start forming a group

of routers. The behavior of the controller when it receives a `JOIN` message depends on the characteristics and requirements of the running applications which are translated and stored on the his memory as network requirements and rules. As soon as the controller receives a number of `JOIN` messages and a group has been created, it broadcasts an `INVITE` message to all participating routers of the group, including a given session participant ID and node ID of all members of the group. The routers afterwards verify that they received the invitation and it follows the group key agreement process, where they are performing two rounds of messages. The two rounds are described in Section 5.2.2.1.

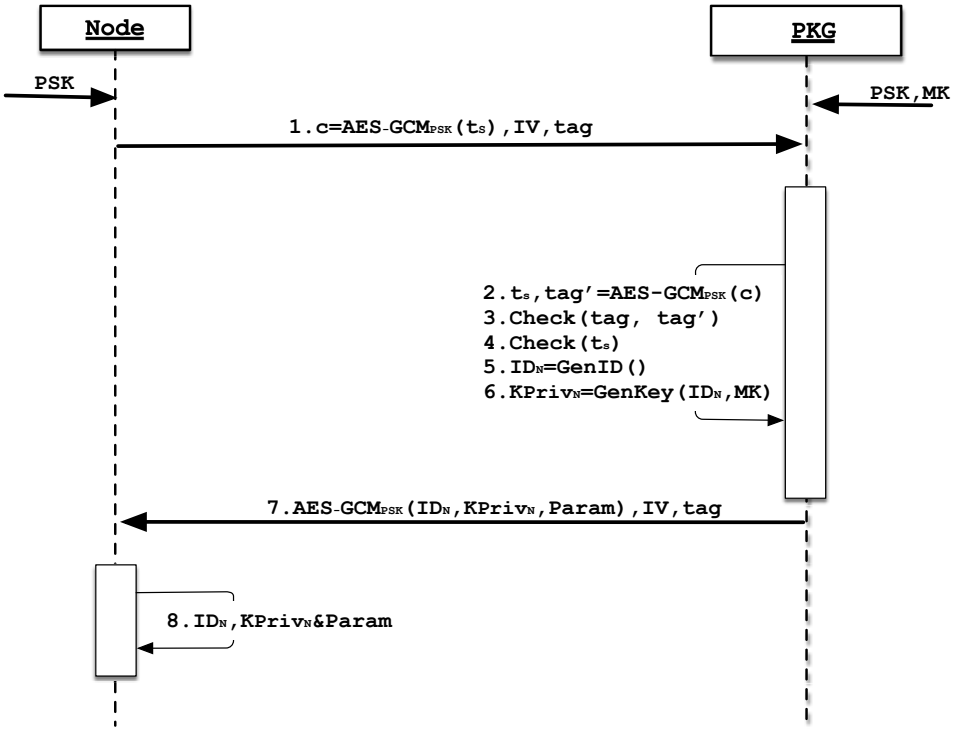


Figure 5.2: Private Key exchange

Phase 3: Router controller communication Once the group has been formed, the last phase can take place, which is used for data exchange. In this phase the controller exchanges data messages with the groups of routers. Furthermore, before the controller starts exchanging any

message with a group of routers, it checks the *Group Table* where the ID's of group participants information is stored, which is described in the previous phase. In case of data transmission between a group of routers and the controller, the controller is using the q secure channel where it encrypts the data. The controller encrypts the data using AES in GCM mode, the group key and an IV.

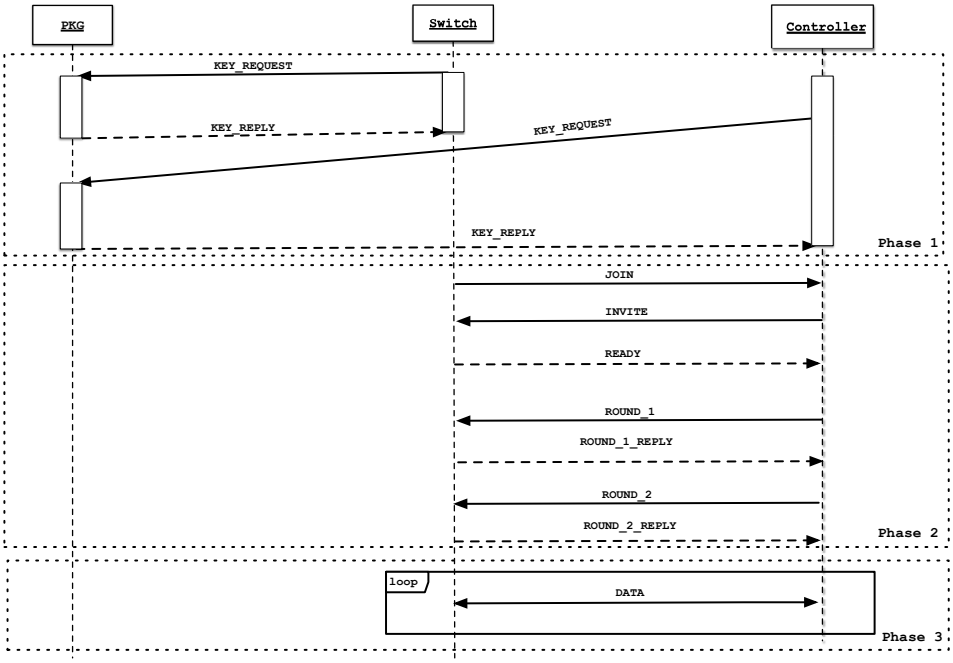


Figure 5.3: SSPSoC message layer.

5.3 Byzantine Faults

Byzantine Fault Tolerance (BFT) is the ability of a network to function as desired and correctly reach a sufficient consensus, despite malicious nodes of the system failing or propagating incorrect information to the other nodes. The design of BFT algorithms originates from the introduction of the Byzantine Generals problem by [LAMPORT et al., 1982], in which the components of a computer system are abstracted as generals of an army. Loyal generals, which are non-faulty components need to find a way to reach to an agreement (e.g. to attack or retreat), while traitors or faulty components are trying to confound others by sending incorrect messages.

In Distributed systems, the communication process and the behavior of networks in the presence of Byzantine faults have been meticulously studied. Interestingly, VLSI circuits can be viewed as Distributed systems at several levels of abstraction: from gates that communicate via binary signals, to components in a NoC. However, the majority of the existing BFT algorithms cannot be implemented within VLSI systems due to the unavailability of the large amount of resources that is required.

The faults can be classified into transient, intermittent, or permanent faults [Constantinescu, 2003]. Regarding SoC, all of the three types of faults can occur in the chip's life cycle. Transient faults appear randomly for one or several cycles. Intermittent faults, which are easily confused with transient faults, occur repeatedly at the same location. They can be tackled by replacing the faulty component hence by removing the fault. Permanent faults can be either logic faults, where transistors or wires are permanently open, or delay faults, where transistors are very slow causing set up and hold timing violation.

Different types of faults, coming from the initial three categories, have been introduced, like crash failures, which are permanent faults occurring when a tile halts prematurely or a link disconnects [Dumitras et al., 2003]. However, arbitrary failures (also called Byzantine), which are transient faults, have not been explored in the context of NoC. Since the NoC has direct access to all communication resources and information flow within the SoC, attackers have a strong motivation to exploit its possible vulnerabilities. Unfortunately, malicious attacks and malicious hardware modifications of a circuit during the design or fabrication process are common causes of failure and they can cause faulty nodes to exhibit arbitrary behavior, that is, Byzantine faults. Malicious attacks that can cause arbitrary faults within NoC are: HT [Bhunia and Tehranipoor, 2018], DoS [Diguet

et al., 2007], HT-DoS [Daoud and Rafla, 2018], etc.

Since there is a big gap in the literature of Byzantine faults in NoC, in this thesis the Byzantine faults are explored and a proposal of a novel security algorithm for tolerating Byzantine faults is presented. This algorithm is specifically designed for a NoC alternative, called SDNoC. SDNoC provides secure paths in presence of untrusted routers and assures that packets will be successfully delivered at their destination. The contribution of this section is summarized as:

- a new fault model, in order to introduce the Byzantine faults within NoC, and
- a novel algorithm relying on SDNoC for tolerating the Byzantine faults.

5.3.1 Related Work

There is no existing literature on BFT algorithms for NoC, however there is a large number of contributions in Distributed systems following Wireless Sensor Network (WSN) and Cloud computing. At the end of the 90s, the pioneers Miguel Castro and Barbara Liskov introduced the practical Byzantine Fault Tolerance (pBFT) algorithm [Castro et al., 1999], which provides a practical Byzantine state machine replication, tolerating malicious nodes within a network by assuming that there are independent node failures and manipulated messages are propagated by specific independent nodes. The algorithm is designed to work in asynchronous systems and can process thousands of requests per second with impressive overhead and a slight increase in latency. However, it is worth mentioning that the communication between the nodes within the system is heavy and each node not only has to prove that the messages came from a specific peer node but additionally needs to verify that the messages were not modified during the transmission.

Following pBFT, several BFT protocols were introduced to improve its robustness, cost and performance [Abd-El-Malek et al., 2005,Cowling et al., 2006, Kotla et al., 2010], while alternative protocols were introduced that leverage trusted components in order to reduce the number of replicas [Chun et al., 2007]. Furthermore, WSNs are prone to Byzantine faults because of their limited energy, low calculation capability and dynamic topology. In [Xu et al., 2015], the authors propose a Byzantine fault-tolerant routing algorithm for large-scale WSN, by ensuring the resistance of timing and energy attacks with help of elliptic curve digital signatures. Afterwards,

in [Panda and Khilar, 2015] a novel distributed fault detection algorithm is presented in order to detect the soft faulty sensor nodes in sparse WSNs, where every sensor node gathers the information only from their neighboring nodes in order to reduce communication overhead.

Cloud-based systems have a more complex architecture in comparison to Distributed systems, they potentially have multiple trust levels and the dynamic change of resources allocated to a service is an easy task in the Cloud. As a result, new BFT algorithms specifically designed for Cloud-based systems have been developed, such as the BFTCloud [Zhang et al., 2011], which is a BFT framework for cloud computing that uses replication techniques to provide the basic fault tolerance and selects voluntary nodes based on QoS characteristics and reliability performance. Another interesting contribution by [Fan et al., 2012], proposes a fault detection strategy for cloud module and cloud application, which can make the cloud application to dynamically detect faults at runtime.

5.3.2 Fault Model

Malicious attacks and malicious hardware modifications of a circuit during the design or fabrication often lead to arbitrary failures and can cause faulty nodes to exhibit arbitrary behavior, these are Byzantine failures. Byzantines failures occur when the system is under specific attacks like HT, DoS, HT-DoS etc.

HT attacks introduce a malicious circuit modification during the design or fabrication process in an untrusted design house or foundry, in which untrusted people, design tools, or components are involved [Bhunia and Tehranipoor, 2018]. Such modifications can lead to abnormal functional behavior of a system, degrade performance and provide covert channels or backdoors by which an attacker can leak sensitive information. More precisely, if a router is infected with a HT, it can maliciously change the flit source or destination address or flit type information of a packet. If a Trojan payload modifies the destination address of a packet, that packet could be directed to an unauthorized IP core.

DoS attacks can make the resources of a system unavailable to legitimate nodes. They can also misroute packets to degrade the network performance causing deadlock and virtual link failure [Daoud and Rafla, 2018].

HT can also launch DoS attacks against the NoC [Zhang et al., 2018b] of a many-core chip by causing serious damages, including dropping of pack-

ets, leaking sensitive information, or modification of functionalities, etc. The consequence of HT-DoS attacks includes bandwidth depletion, incorrect path routing, deadlock and livelock [Diguet et al., 2007].

There is a big number of detection and defense mechanisms specifically designed for each attack separately in literature [Zhang et al., 2018b] [Daoud and Rafla, 2019a], however there is no abstract algorithm that can tackle all these attacks at the same time, ensure the right consensus of the network despite the malicious nodes within the system and preserve the correct functionality of the network.

By taking into account the previously mentioned attacks, this thesis investigates the arbitrary routers by leaving the arbitrary links as a future work. When a NoC is under the above mentioned attacks, the possible arbitrary behavior of a router can include:

- arbitrary deviation from its specification,
- packet redirection,
- packet modification,
- (partial) packet dropping,
- deadlocks or livelocks.

5.3.3 Algorithm

Following the architecture and fault model, the following algorithm was designed, which consists of 2 cases: a) the Normal Case Operation, where the system has no faults and b) the Byzantine fault Case Operation, where the system has faults.

5.3.3.1 Normal Case Operation

The main network entities are the source router, the destination router, the controller and the routers along the route from the source to the destination. The source router is linked to the source PE, which wants to send a packet to a destination PE. The source router will contact the controller, to request a route. Afterwards, the controller, with the help of a routing algorithm described in Section 3.6, will find a route and it will check all the routers along the route for faulty behavior. Thereafter, it will inform the source router for the next hop of the packet. Finally, the source router will await for a final acknowledgment of the packet by the destination in order to

ensure that the packet was successfully delivered. More precisely, each round of the algorithm consists of 6 steps:

- **Step 1:** The source node sends a request to the controller.
- **Step 2:** The controller multicasts the request to the other nodes along the path based on the routing technique that was chosen.
- **Step 3:** The nodes send a reply to the controller.
- **Step 4:** The controller awaits for n replies from the nodes. (n is the number of nodes).
- **Step 5:** The controller sends a message to the source node in order to inform it that the nodes along the path are not faulty and to initiate the routing process.
- **Step 6:** The Destination Node sends an acknowledgment to the source node.

In order to implement the proposed algorithm within NoC, a set of 6 network messages were designed (Table 5.2). Network messages are exchanged between the nodes through physical links following the steps of the Normal Case operation algorithm.

Table 5.2: Designed Network messages

Type	T-Value	Description	Contents
`ROUTE_REQ`	0x01	Sent by source router to controller, which asks a route for a packet.	SRC_ID, DST_ID, Packet_ID, TS
`CONTROL_CHECK`	0x02	Sent by controller to the nodes along the chosen path.	ACK
`CONTROL_REP`	0x03	Sent by the nodes on the path to controller.	NODE_ID, TS
`CONTROL_DONE`	0x04	Sent by controller to the source routers.	PACKET_ID, NEXT_HOP, TS
`ACK`	0x05	Sent by destination router to the source router.	PACKET_ID, TS
`ALERT`	0x05	Sent by source router to the controller in order to inform him that he did not receive an ACK from the destination.	DST_ID, Packet_ID, TS

Figure 5.4 provides an overview of the algorithm, in which the network messages are integrated, in the normal case of no faults. S stands for Source node, C for Controller, N for Nodes along the route and D for Destination node.

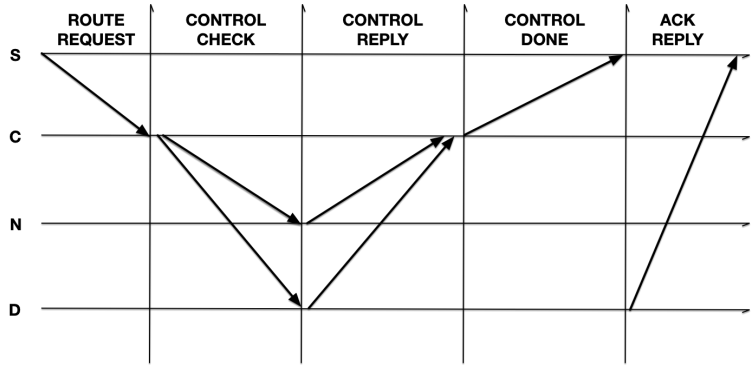


Figure 5.4: Messages under Normal Case operation

5.3.3.2 Byzantine fault Case Operation

In the second scenario, it is considered that the system is equipped with Byzantine faults by following the previously described fault model. In this case, the Normal Case Operation algorithm needs to be enhanced with 2 other algorithms, specifically designed for the controller.

Taking into account the Normal Case Operation algorithm, if a faulty router is present, the first scenario to be considered is that the controller will not receive a reply, `CONTROL_REP`, from the faulty routers along the route. Thus, Algorithm 1 was designed, which is executed by the controller. More precisely, the controller firstly checks if it received a reply from all the routers along the route (*while control_reply[i]==0*, where i is the number of the router). If so, it continues to the next step of the Normal Case algorithm, otherwise it considers the router, from which it did not receive a reply, as a faulty one (*faulty_node = check_node(i)*) and recomputes a new route with the function *new_route()* for the given source and destination of the packet, excluding this router.

The second scenario to be considered is that a faulty router, along the route, could pretend to be legitimate by replying to the controller. However, it sinks the received packet, such that it can never reach its final destination. As a result the destination will not receive any packet and it will not send an `ACK` to the source. Thus, the `ALERT` messages are designed, which are sent from the source to the controller in order to inform that the packet may not have been received by the destination. When the controller receives an `ALERT` message, it initiates Algorithm 2.

Algorithm 1 Faulty Node Algorithm

```
Data: n, source, destination
if control_reply[n] ≠ n then
    for i=1:n do
        while control_reply[i] == 0 do
            faulty_node = check_node(i);
            nroute[]=new_route(source, destination, faulty_node);
            control_check(nroute[]);
        end
    end
else
    control_done();
end
```

Algorithm 2 Alert Algorithm

```
Data: Routes[ ][ ], TrustTable[ ][ ], RouterID, a=0
for k=1:4 do
    if TrustTable[k][2] < tv then
        for j=1:Routes.rows() do
            for t=1:Routes.column() do
                if Routes[j][t] == RouterID then
                    neighbor == Routes[j-1] [t];
                    if TrustTable[k][1] == direction.neighbor() then
                        a=a+1;
                        Suspect[a]=neighbor;
                    end
                end
            end
        end
    end
end
```

Based on the proposed architecture, each router is equipped with a counter in each port (north, east, south, west), which is incremented every time that a new packet is imported and decremented every time that a packet is exported. The results are saved in a *TrustTable*, which includes all the values for the different ports. When the controller receives an `ALERT` message, it requests from all the routers to send their *TrustTable* along with their *RouterID*. The controller calculates and chooses the routes for each individual source-destination pair by storing them in the table *Routes*. The value k indicates the 4 different directions north, east, south, west.

Algorithm 2 is mainly used to identify which are the faulty routers with the help of the table *Suspect*. First the controller checks, whether any input of the *TrustTable* is less than a threshold value (tv). This threshold value can be chosen depending on traffic pattern or buffer holding capacity of the system. If so, then by calculating where this router appears in table *Routes* (*if Routes[j][t] == RouterID*, where j is the row and t is the column of the router), it is searching for the previous hop (*neighbor == Routes[j-1] [t]*, where *j-1, t* are the row and column of the neighbor in the table *Routes*), in order to identify the possible suspect router. Since the controller calculates the table of *Suspect* of the given *RouterID*, it will also check the tables of *Suspect* of the other *RouterID*'s. If a suspect appears at least in two different *Suspect* tables, because each router could have at least two neighbor routers, this router will be considered as faulty.

5.4 Hardware Trojan-Greyhole attack

Following the previous section, a specific Byzantine fault attack was chosen to be investigated. This attack is the HT-DoS. Precisely, in this section a novel HT-DoS attack is introduced.

Malicious hardware modifications at different stages of its life cycle create major security concerns in the field of electronics. The HT attacks emerged as a major security threat for IP blocks, ICs, PCBs, and SoCs. Specifically, these attacks introduce a malicious modification of a circuit during the design or fabrication process in an untrusted design house or foundry, in which untrusted people, design tools, or components are involved [Bhunia and Tehranipoor, 2018]. Such modifications can lead to abnormal functional behavior of a system, degrade performance and provide covert channels or backdoors by which an attacker can leak sensitive information.

According to the literature, HTs have appeared in research around 2005 when the U.S. Department of Defense publicly expressed concerns over the military's reliance on integrated circuits manufactured abroad [Force, 2005]. Furthermore, the fabrication of malicious chips the past years in industrial and military products, made these attacks of bigger concern. In 2010 the chip broker VisionTech was charged with selling fake chips, many of which were destined for safety and security critical systems such as high-speed train breaks, hostile radar tracking in F-16 fighter jets, and ballistic missile control system [Gorman, 2012]. In the future, the threat of hardware Trojans is expected to increase, following the concerns of cyberwar [Smeets, 2018].

Since the number of processors and cores on a single chip is increasing, the interconnection between them becomes significant. A key challenge is to provide secure and reliable communication in the SoC, even in the case that an untrusted NoC IP is inserted into it. Since the NoC has direct access to all communication resources and information flow within the SoC, attackers have a strong motivation to exploit its possible vulnerabilities. In recent literature, a vast number of HT attacks, which mainly focus on NoC, have been introduced [Ancajas et al., 2014, Frey and Yu, 2015, Hussain and Guo, 2017]. Concerning the hardware methods for the detection and defense of the HT-attacks targeting NoC, it is observed that most of them are employed in the NI [Ancajas et al., 2014], which connect the IP cores and routers, some of them on the links between routers [Boraten and Kodi, 2016] and very few on the routers [Frey and Yu, 2015].

Following the literature, the common assumption is that a NoC is supplied to a SoC integrator and there is a possibility that it is already compromised with a HT [Rajesh et al., 2018]. In order to activate the HT, a malicious circuit has already been inserted during the design time of the IP block and a malicious program can activate it later at runtime. The possible attacks due to infected NoC IP block are:

- **Snooping**: In this case, illegal monitoring is performed by an untrusted router within the path, which tracks the number of packets between source and destination IP cores.
- **Corruption of the data**: A malicious router can modify the content of the incoming flits and the route of the packets.
- **Spoofing**: A malicious router copies and replays packets, which may lead to the malfunction or eviction of sensitive data.

- **Denial of service (DoS)**: The denial or distributed denial of service can make the resources unavailable to legitimate PE/routers.

In this thesis, a specific HT assisted DoS attack is considered, called the Greyhole attack, which targets the routers of a NoC within a SoC. The Greyhole attack is a well known attack from WSNs [Tripathi et al., 2013, Martins and Guyennet, 2010]. In case of a Greyhole attack, a malicious router blocks certain packets from its neighboring routers instead of forwarding them. Hence, critical packets, that are forwarded to a Greyhole router, are captured and could not arrive to their destinations. In order to detect and mitigate a malicious router within a network, some of the security mechanisms encountered in the literature are: data partitioning, key management, key generation, localization and trust management [Martins and Guyennet, 2010].

However, despite the large amount of research contributions in WSN about the Greyhole attack, this attack has not been introduced in the field of electronics and more specifically in NoC context. Hence the main contributions are summarized as follows:

- the description and activation of a HT-Greyhole attack in NoC context,
- the exploration of SDNoC as a potential solution for NoCprotection,
- a security management mechanism relying on SDNoC, as key proposal in order to identify malicious routers,
- depending on the position of affected routers, a route exclusion approach is presented in order to mitigate the impact of the attack.

SDNoC provides secure paths in presence of untrusted routers and assures that the packets will be successfully delivered to their destination.

5.4.1 Related work

There is no existing literature on HT-Greyhole attacks, however since Greyhole attacks are variants of Blackhole attacks, the related literature of HT-Blackhole attacks is presented. More precisely, when a blackhole attack is launched, a malicious node stops forwarding or dropping all the incoming packets. In 2018, the HT-Blackhole attack targeting the NoCs was introduced [Zhang et al., 2018b], the authors investigate not only the Blackhole but also the Sinkhole attack in the context of NoC. In a sinkhole attack, the traffic is directed to the hostile node and then many attacks like selective

and blackhole can be empowered by a sinkhole attack. The authors focus on the effects of the attack by measuring the packet loss rate, considering the number of HTs and their distribution in NoC. They provide a theoretical detection method, where a global manager injects detection request packets to randomly selected routers in order to find the suspicious one. Though the main disadvantage of the detection method is that it can only detect HTs which are always on trigger mode. A defense method is also presented, where each router keeps a record of neighbors, which is updated by the global manager and needs to be checked by the router itself before taking routing decisions.

Afterwards, in [Daoud and Rafla, 2019a], an analysis of the HT-Blackhole attack, considering the area and power overhead of the malicious router, was presented. Precisely, the authors presented the influence of the number of HT-Blackhole routers along with their distribution in the NoC. Another contribution by the same authors is presented in [Daoud and Rafla, 2019b], where they proposed a secure protocol with runtime detection and protection of HT-Blackhole attack. The proposed secure protocol protects the system from HT-Blackhole attacks, but it increases dramatically the overhead due to the large number of ACK packets that need to be exchanged between the routers for each data packet transmission from a source to a destination router.

5.4.2 Launching of HT-Greyhole Attack

HTs can be inserted into the pipeline of a VC router according to [Jerger et al., 2017] and at each input port of a router. The main HT is placed on the *VC Allocator* and the other HTs are synchronized with it through a control signal [Zhang et al., 2018b]. A HT structure consists of three modules: the *Trigger*, the *Configuration* and the *Greyhole* function module (Figure 5.5). The placement of the HT-Greyhole in a NoC router is shown in Figure 5.6, where specifically a malicious HT-Greyhole router architecture is illustrated. The router consists of 5 input/output ports, 5 *VC*s, 5 *Buffers* with a counter (C), a *VC allocator*, a *Crossbar switch*, a *Switch allocator*, a *TrustTable* and *Flow Tables*. The five ports correspond to the four cardinal directions and the local direction which connects the router with the PE through the NI. The router consists of a two-stage, pipelined architecture. The first stage is responsible for routing and the second stage is responsible for crossbar traversal. In this work, the functionality of the router is described with respect to a 2D mesh interconnect. A HT is placed in each input port: South, North, East, West, Local, which are synchronized with the main HT-Greyhole which is placed on the *VC allocator*.

More details for the HT insertion in one port in a single cycle VC-based router can be found in [Dimitrakopoulos et al., 2015].

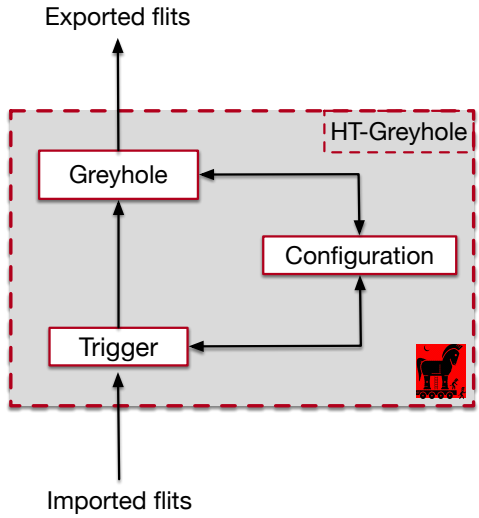


Figure 5.5: HT-Greyhole

Before an attack is launched, a configuration packet should be sent to the target router by an attacker through a malicious program running on the given IP core connected to the router. The configuration packet, which is depicted in Figure 5.7, consists of the following fields:

- **Config cmd**: is the field of a packet that consists of a specific bit pattern (e.g. 00110101), which states as a HT configuration packet.
- **Trigger**: has 2 modes: Always Active (AA) and Destination Based (DB). An AA trigger HT is always active, while a DB trigger is activated only when the destination ID of an incoming packet is identical with the target ID of the configuration packet.
- **Packet Type**: declares the type of packet, which is either a signal or data packet.
- **Activation Signal:** could be on or off depending on the activation of the HT.
- **Target ID**: specifies the target address for the DB trigger.
- **Interceptor ID**: in case that a HT is launched, every data packet *Destination ID* will be replaced with the *Interceptor ID*.

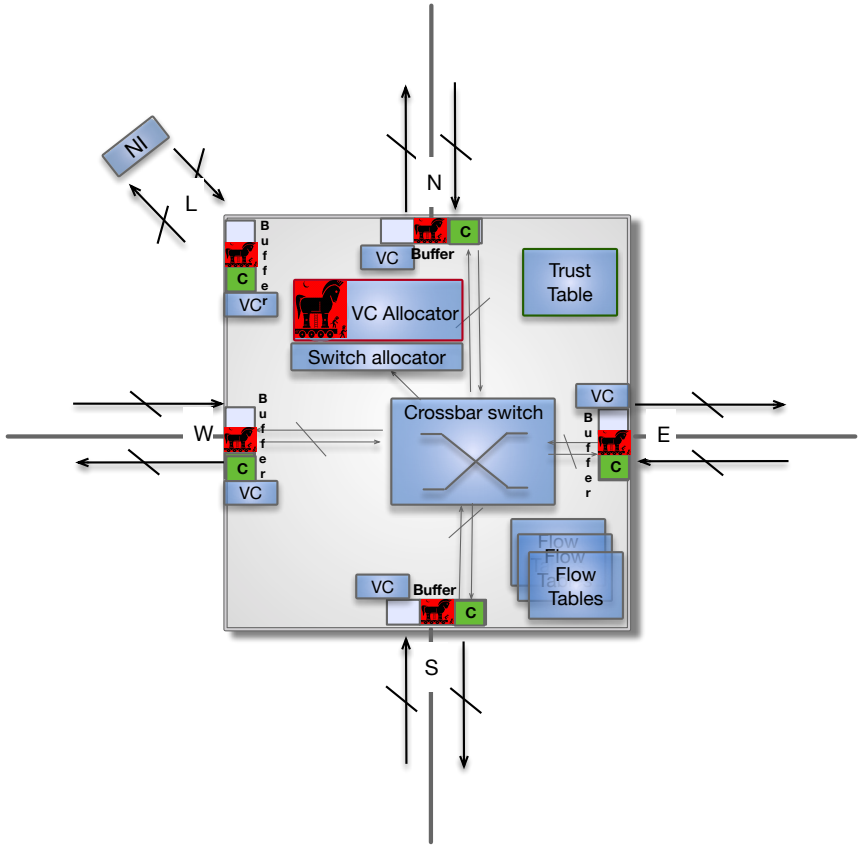


Figure 5.6: HT-Greyhole router.

After the configuration packet has been delivered to the target router, the HT configuration information will be saved in a set of registers (Figure 5.7). When a HT has been configured, it can be activated by the trigger module. More precisely the steps for launching an attack are the following:

- **Step 1**: An attacker sends a configuration packet through a malicious program to the target router.
- **Step 2**: The HT, placed in the target router, receives the configuration packet and updates its configuration information.
- **Step 3**: The trigger module chooses its mode based on the *Trigger* field data stored in the registers.
- **Step 4**: An activation signal is generated by trigger, by taking into account the *Trigger* mode. As for AA mode an activation signal is generated all the time, while for the DB mode the activation signal

is set to on when the *Destination ID* of the incoming packet matches with the *Target ID* of the register.

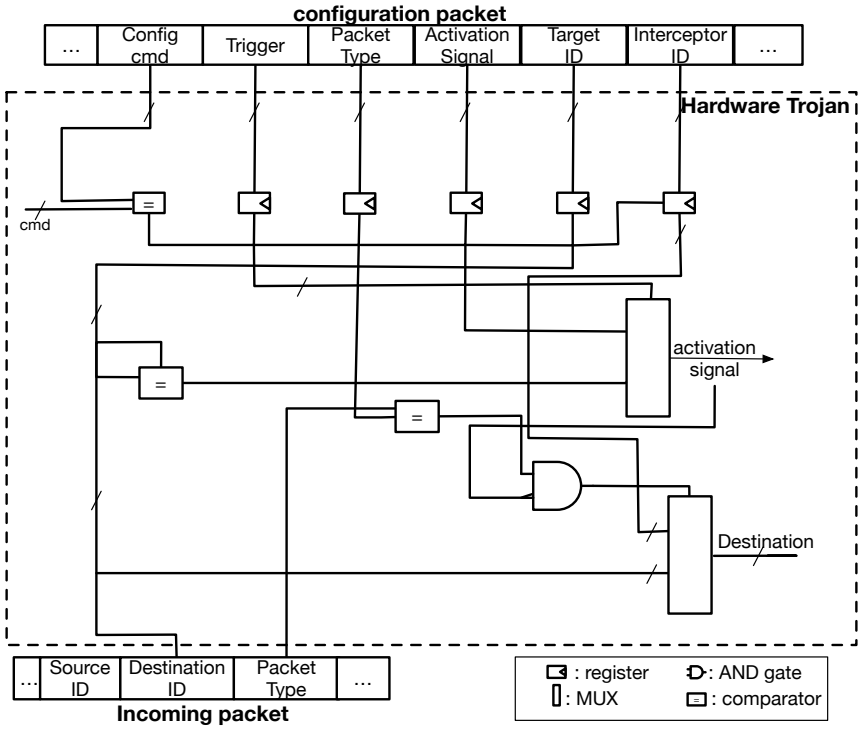


Figure 5.7: HT design on circuit level.

- **Step 5**: The attacker specifies in the configuration packet the type of the packet that needs to be dropped. If the type of the packet matches with the type of the incoming packet then move to the next step (in this scenario the signal packets are normally processed and the data packets are dropped).
- **Step 6**: Launch the attack according to the signal and the *Packet Type*.
- **Step 7**: If the *Packet Type* is *data* then the *Destination ID* of the incoming packet will be replaced with the *Interceptor ID*. If the *Packet Type* is *signal*, the *Destination ID* will not be modified.

5.4.3 Detection

HT assisted DoS attacks are hard to detect due to their low silicon footprint, small power and area consumption but also due to their conditional activation during the run time. Specifically, in the HT attack presented in [Zhang et al., 2018b], the area and the power consumption are 0.07% and 0.02% of a NoC router and in [Daoud and Rafla, 2019a] the malicious router area and power increase are 1.98% and 0.74%, respectively.

The proposed detection strategy has been designed in order to specifically detect a HT-Greyhole attack in the context of SDNoC. Based on the architecture, each router has a counter in each port (Figure 5.6), which is incremented every time that a new packet is imported and it is decremented every time that a packet is exported. The results are saved in the *TrustTable*, which includes all the values for each port. The routers are responsible to periodically send the *TrustTable* along with their *RouterID*, to the controller. The controller calculates and chooses the routes for each individual source and destination by storing them in the table *Routes*. The value k indicates the 4 different directions north, east, south, west.

As soon as the controller receives a *TrustTable*, it uses the Algorithm 3 in order to find out which routers are considered as suspects. In the algorithm, the controller checks if any input of the *TrustTable* is less than a threshold value (tv), which value can be chosen depending on the traffic pattern or buffer holding capacity. More details about the choice of tv value can be found in Chapter 6.4.

Since a malicious router can modify its *TrustTable* and pretend that it is non-malicious, it can only be detected through its neighbors. Hence, the algorithm searches the previous hop (*neighbor*) of the given *RouterID* and afterwards it clarifies if the direction of the *neighbor* matches with the direction of the port value of *RouterID* within the *TrustTable*. If so, the neighbor is considered as suspect. More precisely, different values of tv tested ($0-10$) and according to buffers read and write request of each routers and the traffic pattern, the most suitable threshold was chosen for each scenario.

Since the controller calculates the table *Suspect* of the given *RouterID*, it will also check the tables *Suspect* of the other *RouterID*'s. If a suspect appears at least in two different *Suspect* tables, because each router could have at least two neighbor routers, the suspect router will be considered as malicious.

Algorithm 3 Detection Algorithm

```
Data: Routes[ ][ ], TrustTable[ ][ ], RouterID, a=0
for k=1:4 do
    if TrustTable[k][2] < tv then
        for j=1:Routes.rows() do
            for t=1:Routes.column() do
                if Routes[j][t] == RouterID then
                    neighbor == Routes[j-1] [t];
                     if TrustTable[k][1] == direction.neighbor() then
                        a=a+1;
                         Suspect[a]=neighbor;
                    end
                end
            end
        end
    end
end
```

The detection method is less costly in terms of overhead and complexity since the control links between routers and controller are utilized and the only router side operation is to calculate a *TrustTable*, which includes the values of the 4 counters (4-bit each), and to send it through the control links to the controller.

5.4.4 Defense

As the proposed detection strategy has already identified the malicious routers and their positions, a route exclusion approach is presented in order to mitigate the attack. The controller executes the defense approach which consists of following three phases:

- **Route Exploration Phase:** Given a source and a destination, the controller computes a set of admissible routes based on the OE routing algorithm [Chiu, 2000] and it stores them in a table. OE is a turn model routing algorithm, lightweight and deadlock-free. Among the existing turn model routing algorithms, OE tends to provide better performance and higher adaptiveness than the others.

- **Untrusted Paths Phase:** From the detection algorithm, the controller already has a list with the malicious routers. Hence, in this

phase, it has as input the set of admissible routes from the previous phase, which are checked if they include any malicious router. The routes that include a malicious router are marked as untrusted and the rest of the routes as trusted.

- **Selection Phase:** The inputs in this phase are all the trusted routes from a given source to a destination. In the classic OE routing algorithm, a random route is chosen among the admissible ones. However, in this case the controller chooses the least congested route among the admissible ones by calculating the link load (l_i) of the routes. The l_i corresponds to the number of flits per second that are passing through the link. The designed formula in order to avoid the highly-loaded links and routers within within the route can be computed as:

$$S = \sum_{i=0}^{L_{\mathrm{f}}} l_i. \tag{5.1}$$

Where S is the computed score for each admissible trusted route and L_f is the number of links along the route.

Among the scores of the different routes, the route with smallest score value is selected and indeed it represents the least congested route. Note that the controller's knowledge of the data network state (via the link load) is gathered by immediate inputs from different routers and their *TrustTable* computations. Nevertheless for the initial route computation there is no available score, hence a random route is chosen among the admissible ones, offered by OE.

5.5 Summary

Firstly, a novel GKA communication protocol in order to provide secure communication within a SoC was introduced. The security requirements of the proposed architecture were presented, which includes two phases. In Phase 1, the foremost issue of transporting the PK and the required security parameters of the nodes was addressed and in Phase 2, the router controller communication, where GKA protocols are used, was described. As far as the GKA protocols, two lightweight protocols were chosen and modified in order to fit in the proposed scenario and to be evaluated. Afterwards, the SSPSoC communication protocol was explained, which includes a description of the network architecture, network messages, packet format

and the network initialization. The network initialization consists of three phases: 1) Obtain private key, 2) Form a group and 3) Router controller communication. Following the design of the proposed SSPSoC protocol, its evaluation within an SDN environment is presented in Chapter 6.2.1.

Secondly, a very common problem in all systems was explored, the Byzantine faults. Byzantine faults can cause network performance decrease, higher packet loss and arbitrary behavior of the nodes. However, they remain an unexplored research problem in the context of VLSI systems and more precisely in the NoC. In this thesis an exploration of Byzantine faults within NoC was shown. Precisely, an introduction of a new fault model in NoC context was presented, followed by the design of a novel lightweight algorithm, which includes two cases of operation, and can tolerate Byzantine faults based on SDNoC architecture.

Thirdly, by following the previous research, a specific attack, called HT-DoS, which can cause Byzantine faults, was explored. Specifically, in this thesis the Greyhole attack was introduced. The HT-Greyhole attack targets the routers within NoC by causing performance decrease and packet loss increase. However, during a HT-Greyhole attack, certain packets, which arrive at the router, are dropped, makes it hard to detect. In this chapter, a detection and defense method, against HT-Greyhole attack, which is based on SDNoC architecture were introduced.

As far as the evaluation and implementation of the protocols and algorithms that were introduced in this chapter, will be presented in the following Chapter 6.

Chapter 6

Implementation and Evaluation of security within SDNoC

6.1 Introduction

Following the Chapter 5, in this Chapter an implementation and evaluation of the proposed SSPSoC protocol, the novel Byzantine fault algorithms and a new Hardware Trojan (HT)-Greyhole attack with a defense and a detection mechanism in the context of Software Defined Network-on-Chip (SDNoC) are demonstrated. As far as the implementation is concerned, for the SSPSoC a software Software Defined Network (SDN)-based emulator was used, the Mininet, and for the Byzantine Faults algorithms and HT-Greyhole attack the Garnet Simulator was used, which is described in Chapter 4.2.

6.2 Secure Sdn-based Protocol over mpSoC

6.2.1 Implementation and Performance Analysis

Regarding the performance analysis of the SSPSoC protocol, a simple scenario with three participants: Private Key Generator (PKG), Router and Controller was considered. The messages that will be exchanged between the three participants are depicted in Figure 5.3. As a first step the router and controller will obtain a private key from PKG, by establishing a Transmission Control Protocol (TCP) connection and transmitting the `KEY_REQUEST` message, the PKG will reply with a `KEY_REPLY` message. While a TCP connection is needed in order to conduct a validation of the

proposed protocol, the Layer 4 headers and protocols are not needed in the context on Multi Processor System-on-Chip (MPSoC), thus before its integration into an MPSoC platform some proper modifications should be performed. Afterwards, the Group Key Agreement (GKA) process is executed, where the Sharma protocol and Teng protocol, described in Section 5.2.2.1, were implemented. In order to integrate these two GKA protocols in this scenario, the following steps were implemented:

1. A router broadcasts a `JOIN` message, which contains its ID. The destination is always the controller and it waits for an `INVITE` message.

2. The controller receives the `JOIN` message, makes a decision about the participants of the group (routers) and broadcasts an `INVITE` message to them, which contains the IDs of all the invited participants. Afterwards it waits for `READY` messages.

3. The router receives the `INVITE` message and creates a list of participants. If the ID is valid, it broadcasts a `READY` message, based on the list.

4. The controller remains in idle mode until it receives a `READY` message from the routers that are participants of the group at a specific time. Thereafter, it sends a `ROUND_1` message and waits for `ROUND_1_REPLY` messages.

5. As soon as the router receives the `ROUND_1` message, it broadcasts a `ROUND_1_REPLY` message by waiting for `ROUND_1` and `ROUND_2` messages.

6. When the controller receives the `ROUND_1_REPLY` message from all the participants of the group, it will send `ROUND_2` messages by waiting for `ROUND_2_REPLY` messages.

7. When the controller receives `ROUND_2_REPLY` messages from all routers, the key computation of the group key is started.

8. As a last step the routers, that already belong to a group, can start exchanging `DATA` messages with the controller by using Openflow [McKeown et al., 2008] protocol.

Following the SDN concept, the SSPSoC protocol was evaluated by using the emulator Mininet [Lantz et al., 2010], running on a computer. Mininet is a network emulator, that runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization

to make a single system look like a complete network, running the same kernel, system, and user code. In short, Mininet's virtual hosts, switches, links, and controllers are created using software rather than hardware by having similar behavior to discrete hardware elements.

Concerning the network entities: OpenVSwitches (OVS) [Pfaff et al., 2015] were used as SDNoC routers and a Ryu [Tomonori, 2013] was used as SDNoC controller. The network hosts are emulated using lightweight OS-level visualization: each virtual host inside the mininet network corresponds to a container and it has a virtual network interface with a distinct IP address [Rong and Liu, 2017]. Applications, such as the PKG, controller and node executables can run directly inside virtual hosts. In the experiments, the hosts are interconnected using virtual links and OVS routers are running in kernel mode. In each emulated network instance, one virtual host was used to run the PKG, one host for the controller, and the rest of the hosts to run the nodes participating in the GKA. As far as the implementation of GKA protocols, the PBC [Lynn, 2006] cryptographic library, SHA-256 hash function and AES-GCM cipher are used. Following the PBC library, the Type A (based on symmetric pairing) and Type d159 (based on MNT curves [Miyaji et al., 2001]) parameters were used for the implementation of [Teng and Wu, 2016] and [Sharma et al., 2017] respectively.

6.2.1.1 Network Performance

Simulations were performed for a samples of 1 to 30 nodes (32 virtual hosts in total). Specifically, in order to test the performance of the SSPSoC protocol based on GKA, groups of 2 to 30 nodes (routers) were created. In this research, the first concern was the evaluation of the performance of SSPSoC, by using two different GKA protocols in order to find out which is more appropriate according to their total cost, the cost of `ROUND_1` and `ROUND_2` and the Key Derivation cost. The total cost refers to the time interval between broadcasting a first `INVITE` message and the forming and sending of the first `DATA` message. The cost of two rounds refers to the period between the first `ROUND_1` message or `ROUND_2` message is sent by the controller and the period that the last `ROUND_1_REPLY` or `ROUND_2_REPLY` message is received by the router. The key derivation cost refers to the time that is needed for the group key to be derived (Figure 6.1).

As far as the performance of the controller about Sharma protocol, as depicted in Figure 6.1a, the evolution of the cost is exponential. This appears due to the cost of `ROUND_2` messages, which dominate both the

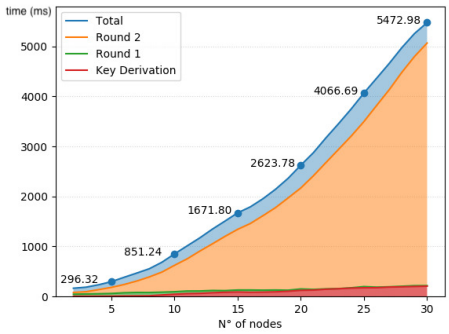
ROUND_1 and the Key Derivation phases concerning its contribution to the total cost. As far as the nodes, as shown in Figure 6.1b, the cost of the protocol still grows exponentially, due to the cost of ROUND_2. It can be noted that the total cost for the nodes is initially higher than the cost of the controller, however it becomes almost equivalent for group sizes of more than 20 nodes. This is due to the implementation, where the controller is among the last participants who generates ROUND_2 messages. In contrast with the first nodes in the group of participants, which receive all the ROUND_2 messages destined to them earlier and thus complete the protocol faster. Therefore, the controller and the average node costs gradually start to become equal due to ROUND_2 dominating the total cost of the protocol.

In both controller and the node cases, the cost of the Teng protocol grows linearly with the size of the nodes, as it was expected. In the case of the nodes, as depicted in Figure 6.1d the ROUND_2 cost is essentially the network cost for the transmission of the ROUND_2 messages, as the calculations are performed by the controller. Similarly the ROUND_1 cost of the controller is the waiting time of ROUND_1 messages from the nodes. Meanwhile, the controller's ROUND_2 cost grows faster that the ROUND_1 cost as the number of calculations depends on the number of participants and eventually dominates the total execution time of the protocol.

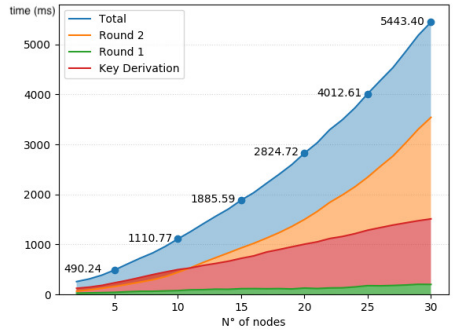
6.2.1.2 Memory Usage

Following the MPSoC concept, another important factor that should be taken into account is the memory usage, since both GKAprotocols are developed for . Hence, scenarios for 5, 10 and 15 nodes were evaluated. The total amount of heap memory allocated during the execution of the SSP-SoC protocol by using the two GKA schemes was measured with Valgrind tool Suite [Nethercote and Seward, 2007], which perform a dynamic binary analysis and enables the Massif heap profiler. The results are presented in Figure 6.2.

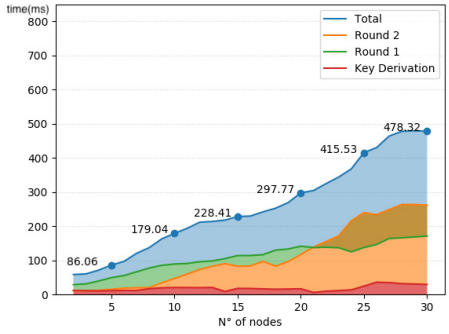
The Sharma protocol (Figure 6.2a) the controller and the nodes use almost the same amount of heap memory, as we would expect from a balanced protocol. In contrast to the Teng protocol (Figure 6.2b), the nodes use about 30% less memory than the controller, which is in line with expectations from an imbalanced protocol. For both protocols, the memory consumption seems to be growing linearly with the number of participants.



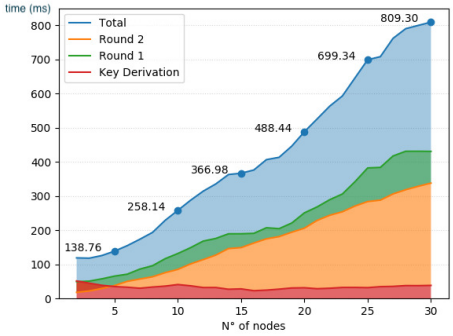
(a) Scalabilty: Controller delay according to Sharma Protocol



(b) Scalabilty: Nodes delay according to Sharma Ptotocol

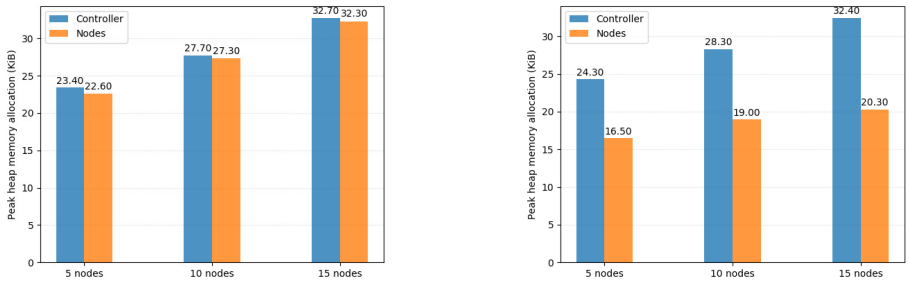


(c) Scalabilty: Controller delay according to Teng Protocol



(d) Scalabilty: Nodes delay according to Teng Protocol

Figure 6.1: Performance results of SSPSoC protocol



(a) Sharma Protocol **(b)** TENG Protocol

Figure 6.2: Memory usage of two GKA protocols

6.2.2 Conclusion

In this research, a new communication protocol based on the group key agreement approach and able to address the inside communication of a MPSoC was proposed. Following the design of the proposed SSPSoC protocol, it was validated and simulated within an SDN environment. The results focused on the evaluation of two GKA schemes according to their scalability and their memory usage. The two main factors making the scalability of Sharma protocol poor are the cost of the scalar multiplications needed for the Round 2 and Signature verification calculations (Chapter 5.2.2.1) and the network cost of Round 2 communications. In addition, it should be considered that both cost factors scale with the number of nodes participating in the protocol, contrary to the number of pairing calculations in the Teng protocol which is fixed. Certainly, the results for the Teng protocol concern Type A pairing parameters, which are the fastest available in the PBC library. However, because only two pairing calculations are involved in the Teng protocol and the message cost scales linearly, while the number of scalar multiplications grows with the number of participants in the Sharma protocol and the message cost grows exponentially, we expect the Teng protocol to be faster for greater number of nodes, regardless of the type of the pairing used. To conclude, the Teng protocol has far better performance and significantly lower power consumption based on the number of participants, making it a more appropriate option for integration in the third phase of the SSPSoC protocol. On the other hand, the Sharma protocol, even without using pairing as Teng protocol, has higher cost and memory usage. These results were obtained due to the authenticity of every participant that the Sharma protocol is considering in contrast to the Teng protocol which does not consider the authenticity of the group participants.

6.3 Byzantine Faults

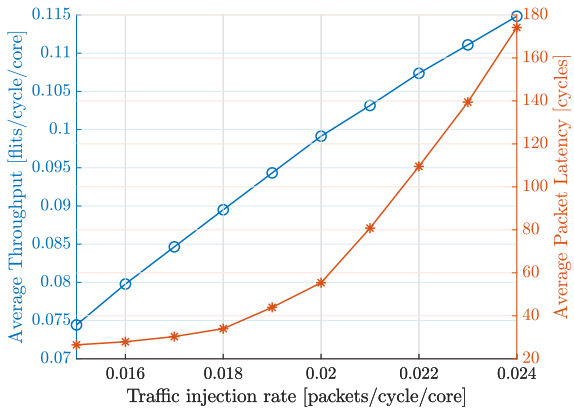
6.3.1 Implementation

Following Chapter 5.3, in order to show how a Byzantine fault can affect the SDNoC and also the improvement of throughput and packet loss, that is accomplished as a result to the proposed algorithms, simulations were performed with Garnet2.0 [Agarwal et al., 2009]. Precisely, the SDNoC architecture was simulated as discussed in Chapter 4 by implementing and evaluating different scenarios in order to explore the effect of Byzantine faults in the network, but also in order to test the proposed algorithms for the correct function of the system. The first scenario represents the Normal Case Operation which is described in Section 5.3.3.1. Afterwards, various scenarios were implemented, in which 1, 3 and 6 Byzantines faults, were imported within the SDNoC and the Byzantine fault Operation algorithms (5.3.3.2) were tested.

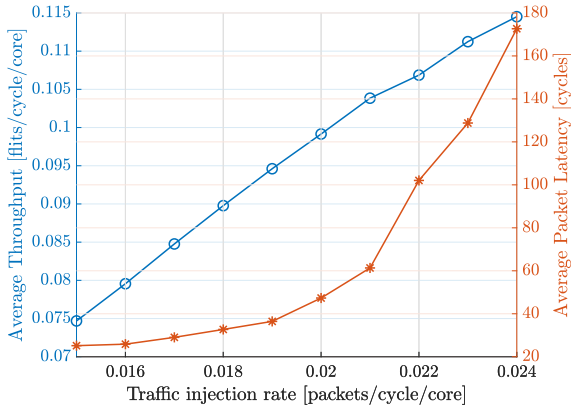
For this scenario an 8×8 topology is simulated, including 0, 1, 3 and 6 Byzantine faults within the network. Furthermore, three different traffic scenarios have been evaluated: Transpose, Uniform and Bit-Reverse. It should be noted that for each scenario 40 iterations are performed, of which the average value of throughput and latency are calculated.

6.3.2 Evaluation

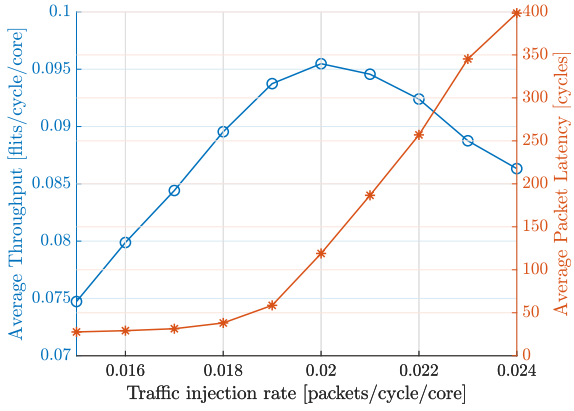
The results of the first scenario, which represents the Normal Case Operation of the proposed algorithm, are depicted in Figure 6.3a, Figure 6.3b, Figure 6.3c. More precisely, in the figures the average throughput and the average packet latency, under different injection rates $(0.015 - 0.024)$, are presented for Transpose, Uniform and BitReverse traffic respectively. The average throughput and latency tend to be identical for Transpose and Bit-Reverse traffic. The average throughput is in the range of $0.075 - 0.115$ flits/cycle/core and the average latency is between $20 - 180$ cycles. As a result, the controller relies on an accurate view of the network state and is able to balance the traffic across the network by avoiding the creation of congested network areas. However, under Uniform traffic the controller is unable to balance the traffic under high injection rate because the source-destination pair is randomly chosen. Hence, in conjunction with the routing algorithm restrictions applied to the routes, the average latency is in the range of $0 - 400$ cycles and the average throughput in the range of 0.0075-0.0095 flits/cycle/core.



(a) Throughput and Latency under Transpose Traffic.



(b) Throughput and Latency under BitReverse Traffic.



(c) Throughput and Latency under Uniform Traffic.

Figure 6.3: Normal Case Operation Scenario measurements.

Table 6.1: Packet loss improvement.

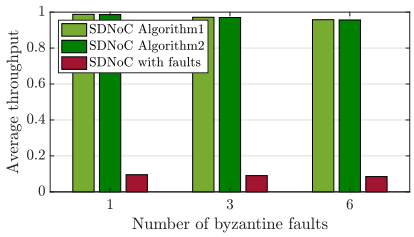
# Byzantine Faults	1		3		6	
# Algorithm	1	2	1	2	1	2
Transpose traffic	24%	15%	56%	47%	76%	65%
BitReverse traffic	24%	14%	55%	46%	77%	67%
Uniform traffic	19%	10%	50%	42%	66%	55%

In Figure 6.4, three different scenarios are presented in each graph. In the first scenario, a single Byzantine fault is imported, the second scenario considers three Byzantine faults and in the third instance, there are six Byzantine faults. Figures 6.4a, 6.4b and 6.4c depict the normalized average throughput under Transpose, Uniform and BitReverse traffic respectively. Figures 6.4d, 6.4e and 6.4f show the normalized packet loss rate. Figure 6.4g, 6.4h and 6.4i illustrate the normalized packet loss. By taking into account these results, the packet loss improvement is shown in Table 6.1. As far as the throughput is concerned, it shows improvement between $62 - 64\%$ for Uniform traffic and $87 - 89\%$ for Transpose and BitReverse traffics. However, with the implementation of the algorithms within the system, there is an increase in the functionalities of the network and hence, there is also a latency increase between 10% and 40%.

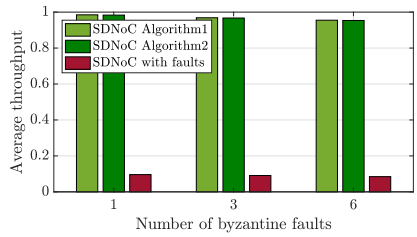
6.3.3 Conclusion

Byzantine faults are a common problem in all systems and can cause network performance decrease, higher packet loss and arbitrary behavior of the nodes. However, they remain an unexplored research problem in the context of VLSI systems and more precisely in the NoC. In this research the Byzantine faults were explored in the context of NoC together with a new fault model, which covers the NoC context. Additionally, a design and evaluation of a novel lightweight algorithm was presented, which includes two cases of operation, and can tolerate Byzantine faults based on SDNoC architecture.

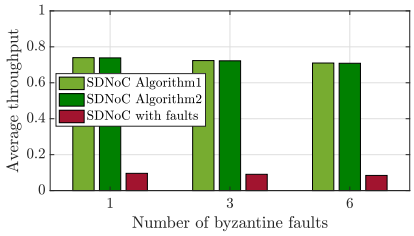
From the results, it is obvious that there is a large throughput decrease and packet loss increase due to the Byzantine faults. Hence, two different algorithms were proposed in order to deal with the reverse arbitrary behavior of the Byzantine fault routers. By applying the proposed algorithms, the NoC continues to function normally by improving the overall packet loss by 23%-77% and the average throughput by 62%-89%.



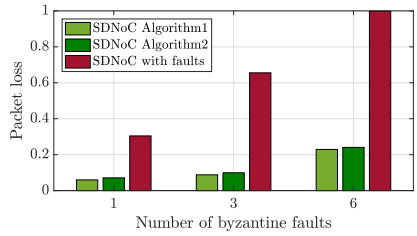
(a) Throughput under Transpose Traffic.



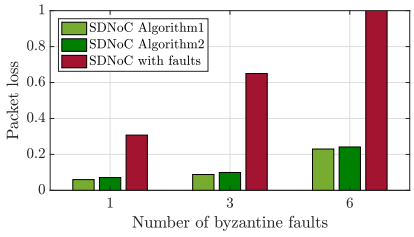
(b) Throughput under BitReverse Traffic.



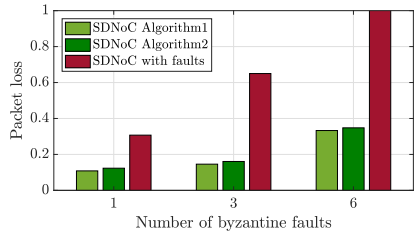
(c) Throughput under Uniform Traffic.



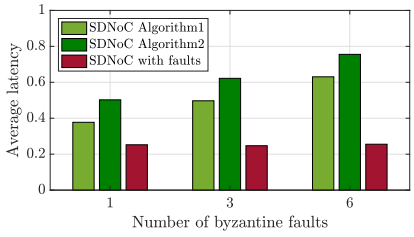
(d) Packet loss under Transpose Traffic.



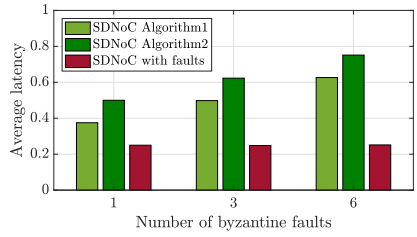
(e) Packet loss under BitReverse Traffic.



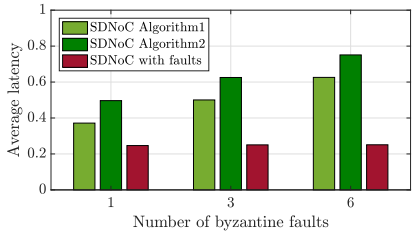
(f) Packet loss under Uniform Traffic.



(g) Latency under Transpose Traffic.



(h) Latency under BitReverse Traffic.



(i) Latency under Uniform Traffic.

Figure 6.4: Byzantine fault case operation scenarios measurements.

The main goal was to achieve the right consensus of the system and the delivery of the packet from the source to the destination. Furthermore, by using the SDNoC architecture, the authenticity of the network is ensured, since there are direct links between the controller and each router. However, the confidentiality and integrity of the network are still open research problems and need further exploration.

6.4 Hardware Trojan-Greyhole attack

Following the Chapter 5.4, the implementation of the HT-Greyhole attack but also the evaluate of the proposed detection and defense strategy were performed. The Garnet2.0 [Agarwal et al., 2009] simulator has been used, which is a NoC model implementation within the gem5 simulator [Binkert et al., 2011]. Precisely, the SDNoC architecture was simulated as discussed in Chapter 4, by implementing the HT-Greyhole attack followed by the detection and defense strategy proposed in Chapter 5.4.3 and Chapter 5.4.4 respectively.

An 8×8 topology is simulated, containing either 1 or 3 or 6 HT-Greyhole routers. Furthermore, three different traffic scenarios have been evaluated: Transpose, Uniform and BitReverse.

6.4.1 Evaluation of the Detection Strategy

6.4.1.1 Background

In order to evaluate the detection algorithm, binary classification is used. Binary or binomial classification is the task of classifying the elements of a given set into two groups (predicting which group each one belongs to) based on the classification rule. Binary classification is the most common classification task.

Data entries $x_1, .., x_n$ have to be assigned into predefined classes $C_1, .., C_l$. In case of binary classification the input classified into one, and only one, of two non-overlapping classes ($C_1 C_2$) [Sokolova and Lapalme, 2009]. Assigned categories can be objective, independent of manual evaluation (e.g, republican or democrat in the votes data of the UCI repository ([Frank, 2010]) or subjective, dependent on manual evaluation (e.g., positive or negative reviews in Amazon.com ([Blitzer et al., 2007]). Classes can be well-defined (e.g., the voting labels), ambiguous (e.g., the review opinion labels), or both.

Table 6.2: Confusion Matrix

		True Condition	
Predicted Condition	***Total Population***	**Positive (P)**	**Negative (N)**
	Positive (P)	True Positive (TP)	False Positive (FP)
	Negative (N)	False Negative (FN)	True Negative (TN)

Considering a two-class prediction problem, in which the outcomes are labeled either as Positive (P) or Negative (N). In this case, there are four possible outcomes from a binary classifier. If the outcome from a prediction is P and the actual value is also P, then it is called a True Positive (TP); however if the actual value is N then it is said to be a False Positive (FP). Conversely, a True Negative (TN) has occurred when both the prediction outcome and the actual value are N and False Negative (FN) is when the prediction outcome is N, while the actual value is P. Based on these parameters, a confusion matrix ca be defined in Table 6.2. The correctness of a classification can be evaluated by computing the number of correctly recognized class examples (TP), the number of correctly recognized examples that do not belong to the class (TN), and examples that either were incorrectly assigned to the class (FP) or that were not recognized as class examples (FN).

Table 6.3 presents the most used measures for binary classification based on the values of confusion matrix. Two of the most used measures are the Sensitivity and the Specificity. Sensitivity, which also called the True Positive Rate (TPR) or Recall, measures the proportion of actual Positive that are correctly identified as Positive. On the other hand Specificity, called the True Negative Rate (TNR), measures the proportion of actual Negative that are correctly identified as Negative. The Positive Predicted Value (PPV) and Negative Predicted Value (NPV) are the proportions of positive and negative results in statistics and diagnostic tests that are TP and TN results, respectively. As far as the Accuracy (ACC) is concerned, it can be described as the degree of closeness of measurements of a quantity to that quantity's true value.

Based on the confusion matrix, a Receiver Operating Characteristic (ROC) graph is a technique for visualizing, organizing and selecting classifiers based on their performance [Fawcett, 2006]. In other words a ROC curve is a graphical representation plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. ROC graphs are two-dimensional graphs in which the TPR is plotted on the Y axis and FPR is plotted on the X axis. A ROC graph depicts the relative trade-off between benefits (TP) and costs (FP). Figure 6.5 shows a ROC graph with five classifiers labeled A through E.

Table 6.3: Measures for binary classification

Measure	**Abr**	**Formula**	**Explanation**
Sensitivity, Recall or True Positive Rate	TPR	$\frac{TP}{TP+FN}$	Effectiveness of a classifier to identify positive labels.
Specificity, Selectivity or True Negative Rate	TNR	$\frac{TN}{TN+FP}$	Effectiveness of a classifier to identify negative labels.
Precision or Positive Predicted Rate	PPV	$\frac{TP}{TP+FP}$	Class agreement of the data with the positive labels given by the classifier.
Negative Predicted Value	NPV	$\frac{TN}{TN+FN}$	Class agreement of the data with the negative labels given by the classifier.
Miss rate or False Negative Rate	FNR	$\frac{FN}{FN+TP}$	Probability of identifying positive labels as negative.
False Positive Rate	FPR	$\frac{FP}{FP+TN}$	Probability of falsely identifying negative labels as positive.
False Discovery Rate	FDR	$\frac{FP}{FP+TP}$	Control of the expected discovered labels that are negative.
False Omission Rate	FOR	$\frac{FN}{FN+TN}$	Compliment of NPV.
Accuracy	ACC	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall effectiveness of a classifier.

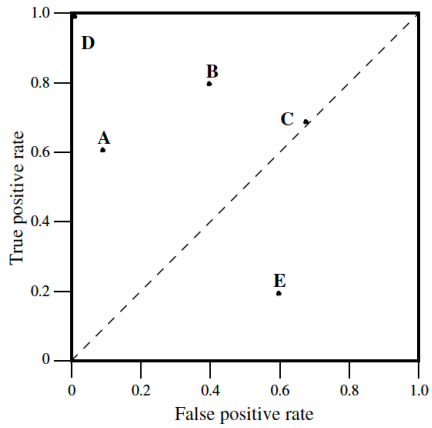


Figure 6.5: ROC space and plots of five discrete classifier [Fawcett, 2006].

The best possible prediction method would yield a point in the upper left corner or coordinates (0,1) of the ROC space, representing 100% sensitivity (no FN) and 100% specificity (no FP). The (0,1) point is also called a perfect classification. A random guess would give a point along a diagonal line from the left bottom to the top right corners (regardless of the positive and negative base rates). The diagonal divides the ROC space. Points above the diagonal represent good classification results and points below the line represent bad classification. As far as the Figure 6.5, D's performance is the best since it lies on (0,1), followed by the performance of A and B. The performance of C lies on random guess line and the performance of E is the worst.

When using normalized units, the Area Under the Curve (AUC) is equal to the probability that a classifier will rank a randomly chosen Positive instance higher than a randomly chosen Negative one [Fawcett, 2006]. In other words the AUC represents the degree or measure of separability. The AUC measure gives a better view about the algorithm's capability of distinguishing between classes. The higher the AUC, the better the model is at predicting the Positive and Negative values. An excellent model has AUC close to the 1, which means it has good separability. A poor model has AUC close to the 0 which means it has bad separability. The formula for the AUC is the following:

$$AUC = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) \tag{6.1}$$

6.4.1.2 Test Cases

By using binary classification, 27 different scenarios of an 8x8 topology were identified, taking into account, different traffics (Transpose, BitReverse, Uniform), different numbers of HT (1, 3, 6) and different threshold (th) values (0, -10, -100) of the detection algorithm. The results of the different scenarios are presented in Table 6.4. For this scenario a malicious node (HT) is considered as Negative and a non-malicious node is considered as Positive, in that setting:

- **TP**: Non-malicious node correctly identified as non-malicious.
- **FP**: Malicious node incorrectly identified as non-malicious.
- **TN**: Malicious node correctly identified as malicious.
- **FN**: Non-Malicious node incorrectly identified as malicious.

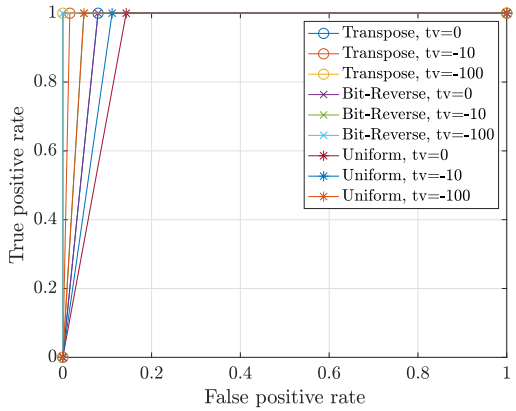
Table 6.4: Results of binary classification for detection algorithm

Traffic	# HT	tv	TP	FN	FP	TN	TPR	FNR	TNR	FPR	PPV	FDR	NPV	FOR	ACC
Transpose	1	0	58	5	0	1	92.1%	7.9%	100%	0%	100%	0%	16.6%	83.3%	92.2%
Transpose	1	-10	62	1	0	1	98.4%	1.6%	100%	0%	100%	0%	50%	50%	98.4%
Transpose	1	-100	63	0	0	1	100%	0%	100%	0%	100%	0%	100%	0%	100%
Transpose	3	0	54	7	0	3	88.5%	11.5%	100%	0%	100%	0%	30%	70%	89.1%
Transpose	3	-10	56	5	0	3	91.8%	8.2%	100%	0%	100%	0%	37.5%	62.5%	92.2%
Transpose	3	-100	60	1	0	3	98.4%	1.6%	100%	0%	100%	0%	75%	62.5%	98.4%
Transpose	6	0	50	8	1	5	86.2%	13.8%	83.3%	16.7%	98%	2%	38.5%	61.%	84.9%
Transpose	6	-10	52	6	0	6	89.7%	10.3%	100%	0%	100%	0%	50%	50%	90.6%
Transpose	6	-100	57	1	0	6	98.3%	1.7%	100%	0%	100%	0%	85.7%	14.3%	98.4%
BitReverse	1	0	57	6	0	1	90.5%	9.5%	100%	0%	100%	0%	14.3%	85.7%	90.6%
BitReverse	1	-10	60	3	0	1	95.2%	4.8%	100%	0%	100%	0%	25%	75%	95.3%
BitReverse	1	-100	63	0	0	1	100%	0%	100%	0%	100%	0%	100%	0%	100%
BitReverse	3	0	52	9	0	1	85.2%	14.8%	100%	0%	100%	0%	25%	75%	85.9%
BitReverse	3	-10	55	6	0	1	90.2%	9.8%	100%	0%	100%	0%	33.3%	66.7%	90.6%
BitReverse	3	-100	61	0	0	3	100%	0%	100%	0%	100%	0%	100%	0%	100%
BitReverse	6	0	48	10	1	5	82.9%	17.2%	83.3%	16.7%	98%	2%	33.3%	66.7%	82.8%
BitReverse	6	-10	53	5	1	5	91.4%	8.6%	83.3%	16.7%	98.1%	1.9%	50%	50%	90.6%
BitReverse	6	-100	58	0	0	6	100%	0%	100%	0%	100%	0%	100%	0%	100%
Uniform	1	0	54	9	0	1	85.7%	14.3%	100%	0%	100%	0%	10%	90%	85.9%
Uniform	1	-10	56	7	0	1	89.9%	11.1%	100%	0%	100%	0%	12.5%	87.5%	89.1%
Uniform	1	-100	60	3	0	1	95.2%	4.8%	100%	0%	100%	0%	25%	75%	95.3%
Uniform	3	0	49	12	0	3	80.3%	19.7%	100%	0%	100%	0%	20%	80%	81.2%
Uniform	3	-10	54	7	0	3	88.5%	11.5%	100%	0%	100%	0%	30%	70%	89.1%
Uniform	3	-100	58	3	0	3	95.1%	4.9%	100%	0%	100%	0%	50%	50%	95.3%
Uniform	6	0	42	16	1	5	72.4%	27.6%	83.3%	16.7%	97.7%	2.3%	23.8%	76.2%	73.4%
Uniform	6	-10	47	11	1	5	81%	19%	83.3%	16.7%	97.9%	2.1%	31.2%	68.8%	81.2%
Uniform	6	-100	54	4	0	6	93.1%	6.9%	100%	0%	100%	0%	60%	40%	93.8%

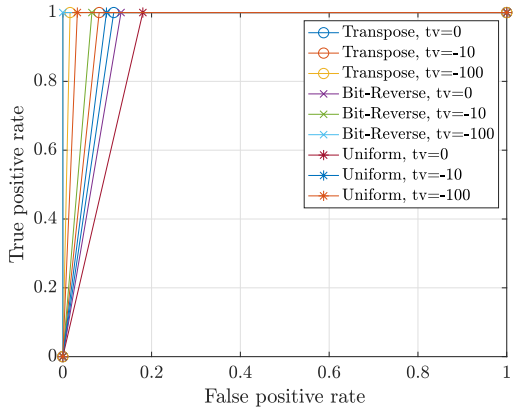
In the test case of 1 malicious node, taking into account the different traffics and for threshold value 0 the TPR values are between 85.7% and 92.1%, the TNR is 100% and the ACC is between 85.9% and 92.2%. In the test case of 3 malicious node, taking into account the different traffics and for threshold value 0 the TPR values are between 80.3% and 88.5%, the TNR is 100% and the ACC is between 81.2% and 89.2%. In the test case of 6 malicious node, taking into account the different traffics and for threshold value 0 the TPR values are between 72.7% and 92.1%, the TNR is 83.3% and the ACC is between 73.4% and 92.2%. Furthermore, it worths to be mentioned that for the last test case for first time 1 FN value is noticed, hence the TNR is lower in contrast to the other test cases.

In the test case of 1 malicious node, taking into account the different traffics and for threshold value -10 the TPR values are between 89.9% and 98.4%, the TNR is 100% and the ACC is between 89.1% and 98.4%. In the test case of 3 malicious node, taking into account the different traffics and for threshold value -10 the TPR values are between 88.5% and 91.8%, the TNR is 100% and the ACC is between 89.1% and 92.2%. In the test case of 6 malicious node, taking into account the different traffics and for threshold value -10 the TPR values are between 81% and 91.4%, the TNR is between 83.3% and 100% and the ACC is between 81.2% and 90.6%. However in the previous test cases the FN value was 1, hence the TNR is lower in contrast to the other test cases, for the 3 different traffic scenarios, in this scenario it is only for Bit-Reverse and Uniform traffic.

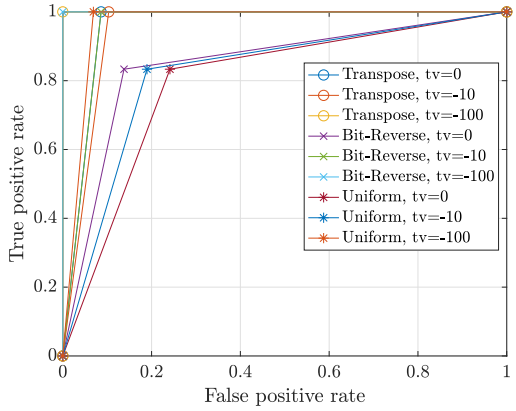
In the test case of 1 malicious node, taking into account the different traffics and for threshold value -100 the TPR values are between 95.2% and 100%, the TNR is 100% and the ACC is between 95.9% and 100%. In the test case of 3 malicious node, taking into account the different traffics and for threshold value -100 the TPR values are between 95.1% and 100%, the TNR is 100% and the ACC is between 95.3% and 100%. In the test case of 6 malicious node, taking into account the different traffics and for threshold value -100 the TPR values are between 93.1% and 100%, the TNR is 100% and the ACC is between 93.8% and 100%. It is obvious that the performance and the accuracy of the algorithm under the threshold value -100 is better for all traffics. Hence the threshold value -100 was consider for the rest of the test cases.



(a) 1 HT-Greyhole router.



(b) 3 HT-Greyhole router.



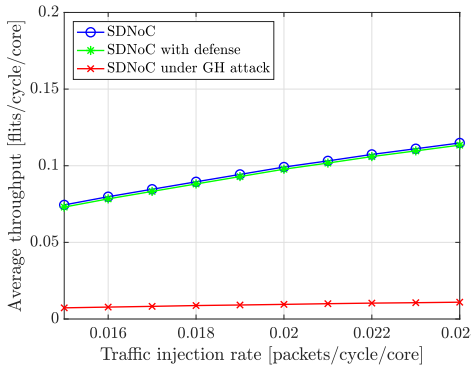
(c) 6 HT-Greyhole router

Figure 6.6: Roc curve diagrams for 1, 3, 6 HT-Greyhole routers with tv=0, -10, -100 and for Transpose, BitReverse, Uniform traffic.

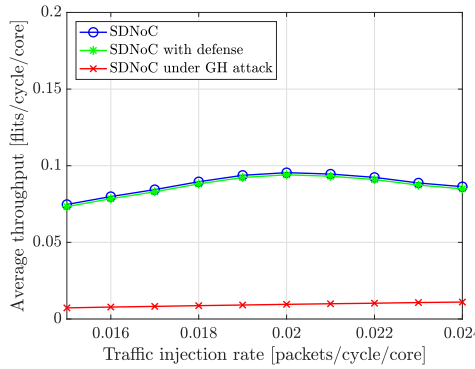
Figure 6.6 represents the ROC curves of the different scenarios for different numbers of HT-Greyhole routers within the network, under different traffic scenarios and different threshold (tv) values. From the graphs, it is obvious that the ROC curves for the $tv = -100$ tend to be ideal for all scenarios. Hence the proposed algorithm is able to better distinguish between positive and negative values for this threshold value. As fas as the ACC of the algorithm is concerned, for $tv = 0$ the ACC is between 73.4% and 92.2%, for $tv = -10$ the ACC is between 81.2% and 98.4% and for $tv = -100$ the ACC is between 95.2% and 100%. However it should be mentioned that for some test cases ($tv = 0$ and $tv = -10$), it is noted that FN values are appeared, hence the FPR will be higher. As far as the AUC value is concerned for $tv = -100$, it is between 0.965 and 1, which means that in some test cases it is perfect (AUC=1) and in other cases it tends to be perfect ($0.95 < AUC < 1$).

Figure 6.7 depicts a scenario of 1 HT-Greyhole router. More precisely, in Figures 6.7a, 6.7b and 6.7c the average throughput under different injection rates (0.015-0.024) is presented for Transpose, Uniform and BitReverse traffic respectively. In Figure 6.7d, 6.7e and 6.7f the packet loss rate is shown under different injection rates. From the figures it is obvious that there is an increase on the packet loss rate and a decrease on the throughput of the SDNoC when the network is under attack compared to when the network works normally. Furthermore, when the proposed defense part is employed on SDNoC, it is noticed that the throughput values of SDNoC with defense and the throughput values of normal SDNoC tend to be identical. Precisely, under the higher injection rate, an increase of 3% is observed under Uniform, Transpose and BitReverse traffics of the overall packet loss rate between SDNoC and SDNoC under HT-Greyhole attack. Regarding the average throughput, it is decreased by 8% under Uniform traffic, 10% under Transpose and BitReverse traffic. Thus, the detection of this attack is a very difficult process.

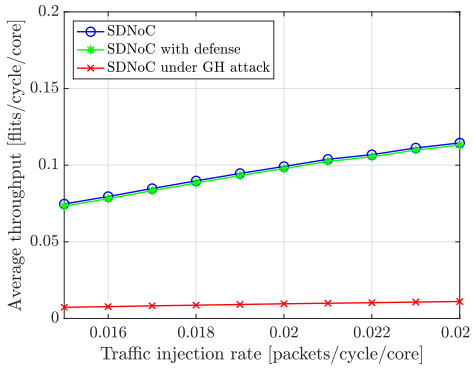
In Figure 6.8, three different scenarios are presented in each graph. In the first scenario, only one HT-Greyhole router is simulated, the second scenario considers three HT-Greyhole routers and in third instance, there are six HT-Greyhole routers. Figures 6.8a, 6.8b and 6.8c depict the *normalized* average throughput under Transpose, Uniform and BitReverse traffic respectively. Figures 6.8d, 6.8e, 6.8f show the *normalized* packet loss rate. By taking into account these results, the packet loss improvement is shown in Table 6.5. As far as the throughput is concerned, by applying the defense method it is improved between 63-66% for Uniform traffic and 88-89% for Transpose and BitReverse traffics.



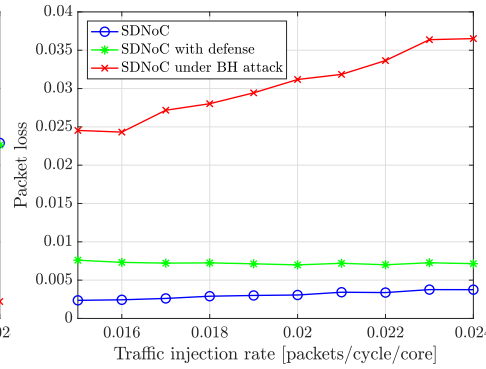
(a) Throughput under Transpose Traffic.



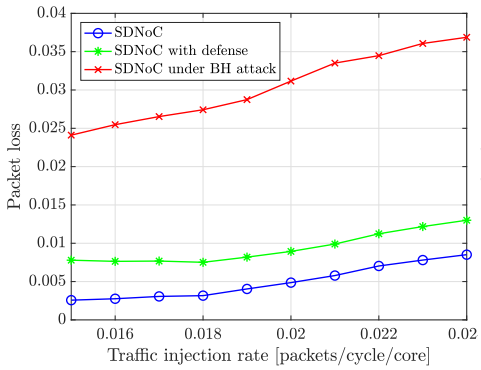
(b) Throughput under Uniform Traffic.



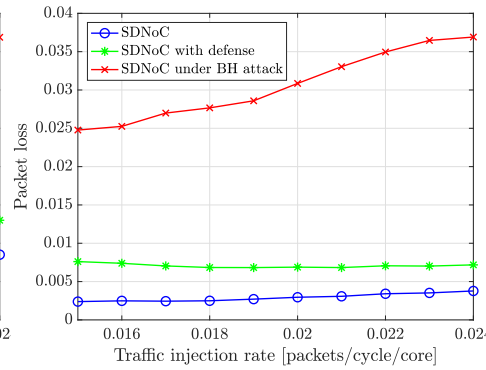
(c) Throughput under BitReverse Traffic.



(d) Packet loss under Transpose Traffic.

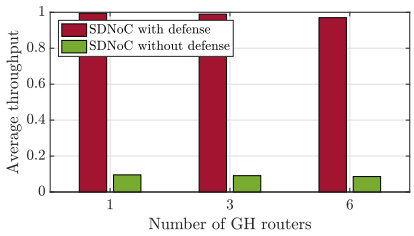


(e) Packet loss under Uniform Traffic.

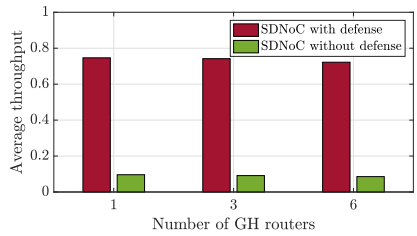


(f) Packet loss under BitReverse Traffic.

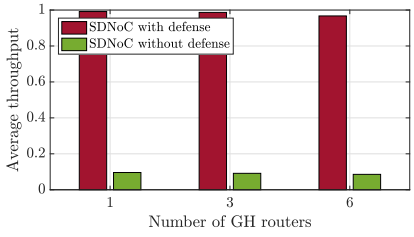
Figure 6.7: 1 HT-Greyhole router under different traffic scenarios.



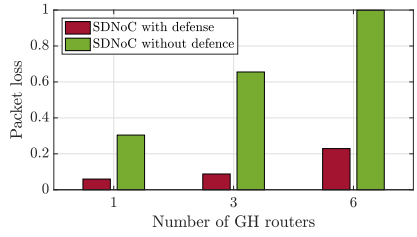
(a) Throughput under Transpose Traffic.



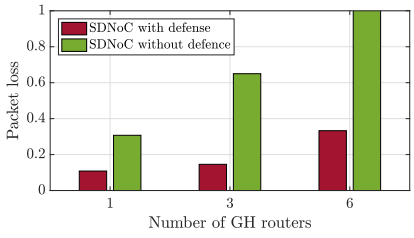
(b) Throughput under Uniform Traffic.



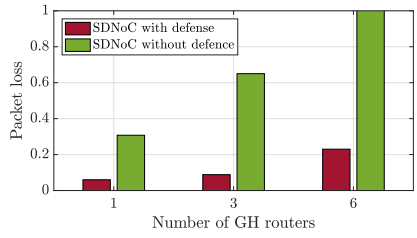
(c) Throughput under BitReverse Traffic.



(d) Packet loss under Transpose Traffic.



(e) Packet loss under Uniform Traffic.



(f) Packet loss under BitReverse Traffic.

Figure 6.8: 1, 3, 6 HT-Greyhole routers scenarios measurements.

Table 6.5: Packet loss improvement with defense method.

# HT-Greyhole Router	**1**	**3**	**6**
Transpose Traffic	27,3%	56,8%	76%
Uniform Traffic	23,6%	50,5%	66%
BitReverse Traffic	27,6%	56,2%	72%

6.4.2 Conclusion

The HT-Greyhole DoS attack targeting NoC can possibly cause network performance decrease and higher packet loss. In this thesis the attack is introduced within SDNoC context and a detection method and a defense method have been designed and evaluated. Through the evaluation of the detection algorithm (using binary classification), the different possibilities of threshold values by finding the most accurate were explored. Afterwards by taking into account the performance results, it is obvious that the packet loss increase and throughput decrease are not significant (3%-10%) enough in order to detect a HT-Greyhole router, due to its stealthy behavior. Hence, the need of an alternate detection method able to detect malicious routers and a defense method which allows the normal function of the systems is mandatory. By applying the defense method, the interconnection system continues to function normally by improving the overall packet loss by 23.6%-77% and the average throughput by 63%-89%.

6.5 Summary-Discussion

This chapter is separated into 3 parts. The first part: the implementation and evaluation of SSPSoC, which is independent from the other two parts: Byzantine faults and HT-Greyhole attack. Precisely, in the first part the implementation and evaluation of a novel SDNoC-based secure GKA communication protocol was presented by evaluating two different GKA within the proposed scenario according to their scalability and their memory usage. From the results it can be noticed that the [Teng and Wu, 2016] protocol outperforms the [Sharma et al., 2017] protocol, hence it was considered as more appropriate in for integration within SSPSoC protocol. It should be noted that the SSPSoC protocol was the first secure communication protocol within SDNoC architecture. However, the protocol has been simulated within a modified software based simulator and in the future it could be tested in a hardware-based simulator for more accurate results.

In the second part, the implementation and evaluation of Byzantine faults in the context of SDNoC were presented. The main objective of this part is the defense against system failures by mitigating the influence of malicious nodes on the correct function of the system and the right consensus that is reached by the legitimate nodes. The proposed algorithms can be used to build highly available NoCs and can tolerate Byzantine faults. Additionally, from the results, it is obvious that when Byzantine faults were implemented within the SDNoC architecture, the throughput was de-

creased and packet loss was increased. Hence, two different algorithms were proposed and evaluated in order to deal with the reverse arbitrary behavior of the Byzantine fault routers. From the evaluation of the proposed algorithms, it was noticed that the NoC continues to function normally by improving the overall packet loss by 23%-77% and the average throughput by 62%-89%. To conclude, the main goal was to achieve the right consensus of the system and the delivery of the packet from the source to the destination. Furthermore, by using the SDNoC architecture, the authenticity of the network is ensured, since there are direct links between the controller and each router. However, the confidentiality and integrity of the network are still open research problems and need further exploration.

In the third part, the implementation and evaluation of a specific Byzantine Fault behavior, that is coming from the HT-Greyhole attack, was presented. Precisely, the HT-Greyhole attack targets the routers within NoC by causing performance decrease and packet loss increase. However, during a HT-Greyhole attack, certain packets, which are arriving towards the router, are dropped which makes it hard to detect. This has been proven through performance results, it is obvious that the packet loss increase and throughput decrease are not significant (3%-10%) enough to detect a HT-Greyhole router, due to its stealthy behavior. Hence, it had been taken into account the accuracy of the proposed detection algorithm. Through the evaluation of the detection algorithm, binary classification was used in order to explore the different possibilities of threshold values by finding the most accurate. As far as the evaluation of the defense method, the interconnection system continues to function normally by improving the overall packet loss by 23.6%-77% and the average throughput by 63%-89%. In the future, more measurements in the context of power and area consumption of this attack could be considered together with the time of HT-Greyhole router detection and its effect on the system.

Chapter 7

Conclusion

The demand for more power-efficient and higher performance computing systems has ushered in the System-on-Chip (SoC) era, where many Intellectual Properties (IP) cores and many processor can be integrated on a single chip. This new trend has provided a higher level of performance for various application requirements. However, as the number of cores, within a single piece of silicon, continuously grows, there is need for scalable on-chip interconnect networks that can deliver high speed data transfer among the many IP cores and processors. According to the literature and recent studies, the traditional interconnects, bus and crossbar, do not scale with an increasing number of cores. Conversely, Network-on-Chip (NoC) has emerged as an alternative and scalable interconnect for future SoCs. However, most existing NoCs suffer from performance degradation due to underutilization of NoC resources. Moreover, it has high complexity and as main communication component between processing core it attracted the attentions of the attackers. Hence researchers, start exploring alternatives of it. The Software Defined Network-on-Chip (SDNoC) is a NoC alternative that gained a lot of attention the last years from the research community. The approach proposed by SDNoC is derived from SDN technology and targets as a main goal the minimization of routers' complexity. Precisely, with the help of SDNoC, the routing logic of the hardware routers, attached in each PE, is exported to a centralized controller, which is running as a software in a given Processing Element (PE). Furthermore, the centralize controller has a general view of the network and can take routing decisions about the packets/flits within the network efficiently.

In this thesis, an attempt has been made towards the research of the the novel SDNoC architecture as a potential solution for future SoCs. Firstly the state of the art of the SDNoC concept was presented. From the presen-

tation of the state of the art, it is obvious that authors focused on different aspects of SDNoC by concentrating mainly on hardware and neglecting the networking functionalities that can be unfolded, but also the security aspects, that can be provided through SDNoC. Hence, in this thesis an effort has been made in order to fill these gaps.

In the context of SDN technology, the OpenFlow Protocol is used for communication between routers and the controller, however its adoptions to micro-scale system is impossible due to its size and design for large scale systems. Following the proposed SDNoC architecture, the first communication protocol called MicroLET [Ellinidou et al., 2019] in the context of micro-scale systems was introduced. The MicroLET protocol is designed in order to provide a new routing approach based on SDN technology and a new message stack specifically designed for micro-scale networks. Furthermore through the evaluation of the MicroLET protocol, it is proven that it could be a good candidate for the future SoCs, as chiplet-based systems. Additionally, in this thesis the routing within SDNoC was explored, where different routing algorithms were implemented and tested within an SDNoC prototype under different scenarios. Also, a new modified version of an already existing routing algorithm was designed and tested in order to obtain better performance results within network. Finally, a statistical analysis was performed in order explore how the performance is affected by the different parameters that has been taken into account within the different simulated scenarios.

As previously mentioned, the security within SDNoC is an unexplored field. Hence, the second field of priority was the investigation of security, by firstly proposing a secure SDNoC-based Group Key Agreement (GKA) protocol, called SSPSoC [Soultana Ellinidou, 2019]. The design of SSPSoC has three main functionalities: the derivation of keys for every node (router or controller) in the network through a Private Key Generator (PKG), the establishment of a secure group of participants, and the secure communication between the participants. Moreover, a simulation and validation of SSPSoC within an SDN environment together with the performance analysis of two GKA protocols, in order to verify which is more suitable in order to cover the second functionality of the proposed protocol in the view of running time and memory consumption were presented.

Afterwards, a common problem within NoC, the arbitrary behavior of routers, Byzantine faults, that can be caused when the system is under different attacks was explored. The Byzantine faults have been very well

investigated in the context of Distributed systems however not in SoCs. Hence, in this thesis a novel fault model followed by the design and implementation of lightweight algorithms, based on SDNoC architecture were proposed [Ellinidou et al., 2020b]. The proposed algorithms can be used to build highly available NoCs and can tolerate Byzantine faults. From the evaluation and simulations of Byzantine faults within a SDNoC, it is obvious that there is a large throughput decrease and packet loss increase. However by applying the proposed algorithms, the SDNoC continues to function normally by improving the overall packet loss by 23%-77% and the average throughput by 62%-89%.

Following the previous contribution, a novel Hardware Trojan (HT)-Denial of Service (DoS) attack, the HT-Greyhole, that causes Byzantine faults was explored. The HT-Greyhole attack is an unexplored attack within NoCs and in this thesis it was implemented and evaluated for first time. Precisely, within this contribution the description and activation of a HT-Greyhole attack in NoC context was presented. Thereafter a security management mechanism relying on SDNoC, as key proposal in order to identify malicious routers and depending on the position of affected routers, a route exclusion approach were presented in order to mitigate the impact of the attack. From the performance results, it was evident that the packet loss was not significantly increased and the throughput was not also significantly decreased (3%-10%). Furthermore, from the evaluation of the defense method, there was a improvement of the overall packet loss by 23.6%-77% and the average throughput by 63%-89%.

Consequently, not only a novel communication and a novel security protocol were proposed but also new possible attacks within SDNoC were explored. The HT-DoS attacks are the new kind of the classic HT attacks that can trick the system and extract sensitive information. For this reason, most of the critical systems need to be specifically designed in order to be able to tackle these kind of attacks during the run time, by detecting them but also defending the system against them. In order to contribute in this field, the HT-Greyhole attack was designed and evaluated. Additionally, with the exploration of Byzantine faults within NoC, a new research field has been opened, where new techniques and algorithms can be designed and proposed.

As fas as the SDNoC integration within the future systems, composed of many nodes, highly configurable communication such as SDNoC seemed to be very promising however more research in the different communica-

tion hierarchical levels need to be done. For example, in the case of CoC architecture [Bousdras et al., 2018] each IC could integrate a software-programmable controller and all the controllers will report to the central hardware controller. The two-level of hierarchy enables efficient communication on the IC level as well as the PCB level. As far as, the packet forwarding it can be managed in the same way that is described in Chapter 3.4. The source IP core forwards the packet header to the controller and the controller sends back the exit port at each router on the path. Furthermore, the controllers on each IC also will maintain flow tables and group tables for outside IC communication. The flow rules include frequently visited paths, and in a case of miss, the packet header is forwarded to the central controller. The central controller has access to global topology view and is responsible for the updates of flow entries on these controllers. Once the flow entry is updated, the header packet is assigned a route and the rest of the packets will follow the same route. As far as the chiplets architecture is concerned, the MicroLET protocol (Chapter 3.6) designed to cover the intra-chiplet communication. In that case the controller will be placed inside a chiplet and attached to one router, the rest of the routers within the network will communicate in order to ask for a possible route for the upcoming packets from the controller. However in the case of inter-chiplet communication, its chiplet may contain its own local sub-SDNoC and an extra SDNoC will be placed on the interposer in order to transfer packets between different chiplets. In this case two-level hierarchy is needed with a main-controller within the interposer, which is managing the sub-controllers within chiplets and is able to transfer packets through different chiplets.

7.1 Future Work

As previously mentioned, the security field constitutes a huge gap in the context of SDNoC. It can be seen that except for network efficiency that the SDNoC technology also brought new malicious attacks that need to be considered. It was already explained in the STRIDE model, in Chapter 3.2.1, which is applicable for SDNs, since a lot of researchers try to address the possible threats that the SDN is posing. However, from NoC and SDNoC point of view, there is not any security model present in literature that addresses the possible attacks of the network. Since SDNoC is a co-design of hardware and software, it brought new threats into the surface, which need to be addressed especially during the design process of a NoC IP and before its integration on a SoC. A detailed analysis of the possible threats due to STRIDE model is presented in the Table 7.1. However,

more research contributions need to be presented in this field by analyzing the possible threats that are posed for the SDNoC but also by proposing structural solutions in order to tackle them and maintain the function of the system. The research community needs to investigate different types of attacks coming from both software and hardware and take into account these attacks during the design of SDNoC IPs.

In order to fully exploit the SDNoC technology, enhance its functionalities and maintain the secure communication of the system in the future, more research investigating on the controller side is expected. The controller is the key element of the SDNoC technology, since it has a broad view of the network and it is able to apply rules and manage the routing efficiently. With the capability of monitoring the network, the controller gathers information about the routers and packets. Hence, this data can be used as input of machine learning algorithms. Based on real-time network data, machine learning techniques can bring intelligence to the controller by performing data analysis, network optimization, and anomaly detection [Xie et al., 2018]. Moreover, thanks to programmability of SDN, optimal network solutions (e.g., configuration and resource allocation) made by machine learning algorithms can be executed on the network in real time. To conclude, in order to exploit the SDNoC technology withing SoCs, supervised machine learning algorithms for the controller should be designed.

In this thesis the controller is considered as a trusted entity, however the controller could be a single point of failure. If the controller fails, the network will cease to function, which significantly reduces the reliability of the NoC. A possible solution could be a back-up controller or the creation of different levels of controllers. Since the processing cores in a SoC are increasing, the future system will not be able to function with only one controller. In that case, the controller will be overloaded and it will consume a major amount of power. Hence, the solution of multiple controllers seems to be ideal. In the case of multiple controllers, some reference architectures have already been introduced [Krishnamurthy et al., 2014] [Phemius et al., 2014] in SDN field. The controllers can form a peer-to-peer, high-speed, reliable and distributed network control. The routers in the infrastructure plane, forward packets among them by checking the flow tables that are controlled by the controller(s) in the control plane. In future SoCs, like CoC or chiplet architectures, the need of multiple controllers can be more obvious due to its hierarchical hardware levels. Hence, more research contributions are expected in this research topic.

Table 7.1: STRIDE Model analysis for SDN and SDNoC

	Attacks	
STRIDE	**SDNoC**	**SDN**
Spoofing	A malicious SDNoC could pretend to be trusted in order to copy and replay packets [Ancajas et al., 2014], [Sepúlveda et al., 2017], [Rajesh et al., 2018]. A compromised SDNoC can spoof any node in order to create a dummy request of privileged information e.g. Spoofed router [Biswas et al., 2015].	A spoofed SDN controller could take the control of the whole network, however a spoofed router could only attack the data routed through it [Hu et al., 2015].
Tampering	An attacker is able to tamper with the NoC IP before its integration into the SoC [Sepúlveda et al., 2017].	This could happen when the controller installs flow rules, aiming to cause system misbehavior like an attack of Fake LLDP Injection [Hong et al., 2015].
Repudiation	-	In this case a controller or a switch can deny to be involved in a communication. Consequently, non-repudiation appears to ensure such denial does not occur. It can be also caused by Man In The Middle (MITM) attacks, by hijacking the channel between two parties and persuading that they are the other party. [Li et al., 2016]
Information disclosure	A possible chip vulnerability can lead to information disclosure. e.g Intel's processors vulnerability (Foreshadow attack [Van Bulck et al., 2018]), which allows any application running on a user–level mode to access protected memory areas, by giving the chance to the attacker to access sensitive data. Also Meltdown [Lipp et al., 2018] and Spectre [Kocher et al., 2019] CPU vulnerabilities, which affected the processors used over the past two decades.	If an attacker reaches the switches he can tamper with flow rule causing the traffic to go to the wrong destination. Information Disclosure could be performed by Man At The End (MATE), if the attacker can gain information, which allows him to log into the system as an administrator and reach the controller [Eldewahi et al., 2018]. MITM is an information disclosure attack that targets also the information in the transit [Brooks and Yang, 2015].
Denial of service	There are three main types of DoS attacks on NoCs [Fiorin et al., 2007]: Bandwidth Reduction [JS et al., 2015] [Ancajas et al., 2014], where frequent and useless packets are inserted in the network in order to waste bandwidth and cause a higher latency in on-chip communications. Draining Attacks, aiming at reducing the operative life of a battery powered device by making the system execute power hungry tasks. Hardware Trojans may cause retransmissions which may lead to a DoS attack by creating false congestion between the routers [Boraten and Kodi, 2016], [Malekpour et al., 2017].	The controller should be aware of the network state on a regular basis in order to apply rules. This makes an SDN based system vulnerable for DoS is possible by flooding the controller-switch communication or the flow tables of a switch [Yan et al., 2016], [Yan and Yu, 2015, Dover, 2013].
Elevation of privilege	Semiconductor giant releases patch for its Intel Active Management Technology vulnerability that could allow an attacker to escalate privileges in its high-end chipset [Intel, 2017].	In order to perform this attack in SDN, an attacker should have access to the controller, which is considered as a less critical to happen, due to the use of TLS [Sezer et al., 2013]. If an attacker gets privileged access, then the entire routing process can be changed which can stop or destroy the entire system.

Appendix

Appendix A

GEM5 Code

```
#include "mem/ruby/network/garnet2.0/RoutingUnit.hh"

#include "base/cast.hh"
#include "mem/ruby/network/garnet2.0/InputUnit.hh"
#include "mem/ruby/network/garnet2.0/Router.hh"
#include "mem/ruby/slicc_interface/Message.hh"

// changes by Soultana Ellinidou-SDNoC
#include "mem/ruby/network/garnet2.0/OutputUnit.hh"
//

RoutingUnit::RoutingUnit(Router *router)
{
    m_router = router;
    m_routing_table.clear();
    m_weight_table.clear();

    // changes by Soultana Ellinidou-SDNoC
    std::ifstream reader;
    reader.open("/home/gaurav/gem5/var/timeout.txt");
    if (!reader)
        assert(0);
    reader >> timeout;
    reader.close();

    std::ifstream reader2;
    reader2.open("/home/gaurav/gem5/var/Kroute.txt");
    if (!reader2)
        assert(0);
    reader2 >> Kroute;
    reader2.close();

    std::ifstream reader3;
    reader3.open("/home/gaurav/gem5/var/beta.txt");
    if (!reader3)
        assert(0);
    reader3 >> beta;
    reader3.close();

    std::ifstream reader4;
    reader4.open("/home/gaurav/gem5/var/gamma.txt");
    if (!reader4)
        assert(0);
    reader4 >> gamma;
    reader4.close();


    std::ifstream reader5;
    reader5.open("/home/gaurav/gem5/var/tau.txt");
    if (!reader5)
        assert(0);
    reader5 >> tau;
    reader5.close();

```

```
55      std::ifstream reader6;
56      reader6.open("/home/gaurav/gem5/var/penality.txt");
57      if (!reader6)
58          assert(0);
59      reader6 >> penality;
60      reader6.close();
61  }
62
63  void
64  RoutingUnit::addRoute(const NetDest& routing_table_entry)
65  {
66      m_routing_table.push_back(routing_table_entry);
67  }
68
69  void
70  RoutingUnit::addWeight(int link_weight)
71  {
72      m_weight_table.push_back(link_weight);
73  }
74
75
76  int
77  RoutingUnit::lookupRoutingTable(int vnet, NetDest msg_destination)
78  {
79      // First find all possible output link candidates
80      // For ordered vnet, just choose the first
81      // (to make sure different packets don't choose different routes)
82      // For unordered vnet, randomly choose any of the links
83      // To have a strict ordering between links, they should be given
84      // different weights in the topology file
85
86      int output_link = -1;
87      int min_weight = INFINITE_;
88      std::vector<int> output_link_candidates;
89      int num_candidates = 0;
90
91      // Identify the minimum weight among the candidate output links
92      for (int link = 0; link < m_routing_table.size(); link++) {
93          if (msg_destination.intersectionIsNotEmpty(m_routing_table[link])) {
94
95              if (m_weight_table[link] <= min_weight)
96                  min_weight = m_weight_table[link];
97              }
98      }
99
100     // Collect all candidate output links with this minimum weight
101     for (int link = 0; link < m_routing_table.size(); link++) {
102         if (msg_destination.intersectionIsNotEmpty(m_routing_table[link])) {
103
104             if (m_weight_table[link] == min_weight) {
105
106                 num_candidates++;
107                 output_link_candidates.push_back(link);
108             }
109         }
110     }
111
112     if (output_link_candidates.size() == 0) {
113         fatal("Fatal Error:: No Route exists from this Router.");
114         exit(0);
115     }
116
117     // Randomly select any candidate output link
118     int candidate = 0;
119     if (!(m_router->get_net_ptr())->isVNetOrdered(vnet))
120         candidate = rand() % num_candidates;
121
122     output_link = output_link_candidates.at(candidate);
123     return output_link;
124 }
125
126
127 void
128 RoutingUnit::addInDirection(PortDirection inport_dirn, int inport_idx)
129 {
130     m_inports_dirn2idx[inport_dirn] = inport_idx;
131     m_inports_idx2dirn[inport_idx]  = inport_dirn;
132 }
133
134 void
```

```
135 RoutingUnit::addOutDirection(PortDirection outport_dirn, int outport_idx)
136 {
137     m_outports_dirn2idx[outport_dirn] = outport_idx;
138     m_outports_idx2dirn[outport_idx]  = outport_dirn;
139 }
140
141 int
142 RoutingUnit::outportCompute(RouteInfo route, int inport, PortDirection
        inport_dirn)
143 {
144     int outport = -1;
145     if (route.dest_router == m_router->get_id()) {
146
147         // Multiple NIs may be connected to this router,
148         // all with output port direction = "Local"
149         // Get exact outport id from table
150         outport = lookupRoutingTable(route.vnet, route.net_dest);
151         return outport;
152     }
153
154     // Routing Algorithm set in GarnetNetwork.py
155     // Can be over-ridden from command line using --routing-algorithm = 1
156     RoutingAlgorithm routing_algorithm =
157         (RoutingAlgorithm) m_router->get_net_ptr()->getRoutingAlgorithm();
158
159     switch (routing_algorithm) {
160         case TABLE_:  outport =
161             lookupRoutingTable(route.vnet, route.net_dest); break;
162         case XY_: outport =
163             outportComputeXY(route, m_router->get_id(), inport_dirn)->get_id();
        break;
164         default: outport =
165             lookupRoutingTable(route.vnet, route.net_dest); break;
166     }
167     assert(outport != -1);
168     return outport;
169 }
170
171 OutputUnit*
172 RoutingUnit::outportComputeXY(RouteInfo route, int current, PortDirection
        inport_dirn)
173 {
174     Router* router = m_router->get_net_ptr()->getRouter(current);
175     PortDirection outport_dirn = "Unknown";
176     OutputUnit* outport;
177
178     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
179     int num_cols = m_router->get_net_ptr()->getNumCols();
180     assert(num_rows > 0 && num_cols > 0);
181
182     //int my_id = m_router->get_id();
183     int my_id = current;
184     int my_x = my_id % num_cols;
185     int my_y = my_id / num_cols;
186
187     int dest_id = route.dest_router;
188     int dest_x = dest_id % num_cols;
189     int dest_y = dest_id / num_cols;
190
191     int x_hops = abs(dest_x - my_x);
192     int y_hops = abs(dest_y - my_y);
193
194     bool x_dirn = (dest_x >= my_x);
195     bool y_dirn = (dest_y >= my_y);
196
197     // already checked that in outportCompute() function
198     assert(!(x_hops == 0 && y_hops == 0));
199
200     if (x_hops > 0) {
201         if (x_dirn) {
202             assert(inport_dirn == "Local" || inport_dirn == "West");
203             outport_dirn = "East";
204         } else {
205             assert(inport_dirn == "Local" || inport_dirn == "East");
206             outport_dirn = "West";
207         }
208     } else if (y_hops > 0) {
209         if (y_dirn) {
210             // "Local" or "South" or "West" or "East"
211             assert(inport_dirn != "North");
```

```
212             outport_dirn = "North";
213         } else {
214             // "Local" or "North" or "West" or "East"
215             assert(inport_dirn != "South");
216             outport_dirn = "South";
217         }
218     } else {
219         // x_hops == 0 and y_hops == 0
220         // this is not possible
221         // already checked that in outportCompute() function
222         assert(0);
223     }
224
225     outport = router->get_map_direction_out(outport_dirn);
226     return outport;
227 }
228
229
230 int
231 RoutingUnit::outportComputeCustom(RouteInfo route, int inport, PortDirection
        inport_dirn)
232 {
233     assert(0);
234     return -1;
235 }
236
237
238 // changes by Soultana Ellinidou-SDNoC
239 /**********************************/
240
241 // SDN controller
242 int
243 RoutingUnit::outportBack(RouteInfo route)
244 {
245     int back_to_node = m_router->get_map_dst_out(route.src_router)->get_id();
246     return back_to_node;
247 }
248
249 int
250 RoutingUnit::outportBack_hack(RouteInfo route)
251 {
252     int back_to_node = m_router->get_map_dst_out(route.dest_router)->get_id();
253     return back_to_node;
254 }
255
256
257 int
258 RoutingUnit::outportController(RouteInfo route)
259 {
260     int outport = -1;
261
262     if (route.dest_router == m_router->get_id()) {
263
264         // Multiple NIs may be connected to this router,
265         // all with output port direction = "Local"
266         // Get exact outport id from table
267         outport = lookupRoutingTable(route.vnet, route.net_dest);
268         return outport;
269     }
270
271     return m_outports_dirn2idx["to_sdn_src"];
272 }
273
274 int
275 RoutingUnit::NoC_outportCompute(RouteInfo route)
276 {
277     int outport = -1;
278
279     if (route.dest_router == m_router->get_id()) {
280
281         // Multiple NIs may be connected to this router,
282         // all with output port direction = "Local"
283         // Get exact outport id from table
284         outport = lookupRoutingTable(route.vnet, route.net_dest);
285         return outport;
286     }
287     return outport;
288 }
289
290 void
```

```
291 RoutingUnit::SDN_outportCompute(RouteInfo route)
292 {
293     Route final_route;
294     Route mroute;
295     std::vector<Route> setRoute;
296 
297     //*****************************
298     // Routing algorithm
299 
300     RoutingAlgorithm routing_algorithm = (RoutingAlgorithm) m_router->
        get_net_ptr()->getRoutingAlgorithm();
301     int current = route.src_router;
302     PortDirection inport_dirn = "Local";
303 
304     if (routing_algorithm == 1)
305         outportComputeCustomXY(route, current, inport_dirn, mroute, setRoute);
306 
307     else if (routing_algorithm == 14)
308         outportComputeCustomOE(route, current, inport_dirn, mroute, setRoute);
309 
310     else if (routing_algorithm == 13)
311         outportComputeCustomNF(route, current, inport_dirn, mroute, setRoute);
312 
313     else if (routing_algorithm == 12)
314     {
315         outportComputeCustomNL(route, current, inport_dirn, mroute, setRoute);
316     }
317 
318     else if (routing_algorithm == 11)
319         outportComputeCustomWF(route, current, inport_dirn, mroute, setRoute);
320 
321     else
322     {
323         outportComputeCustomOE(route, current, inport_dirn, mroute, setRoute);
324     }
325 
326 
327     //*****************************
328     // Selection
329 
330     if (routing_algorithm == 21)
331       final_route = selection_link_max(setRoute, route);
332 
333     else if (routing_algorithm == 22)
334       final_route = selection_link_sum(setRoute, route);
335 
336     else if (routing_algorithm == 31)
337       final_route = selection_router_max(setRoute, route);
338 
339     else if (routing_algorithm == 32)
340       final_route = selection_router_sum(setRoute, route);
341 
342     else
343         final_route = setRoute[rand() % setRoute.size()];
344 
345     //*********************
346     // Update flow tables
347 
348     for (auto elem : final_route)
349     {
350         Router* router = elem->get_router();
351         std::pair<int, int> pair = std::make_pair(route.src_router, route.
        dest_router);
352         router->set_flow_table(pair, elem->get_id());
353         router->set_flow_timeout(pair, curTick() + timeout);
354 
355         // Penality
356         double tmp = penality /(double) tau;
357         InputUnit* in = m_router->get_net_ptr()->get_map_link_inport(elem->
        get_nwk_link());
358         in->set_state_rate_saved(in->get_state_rate_saved() + tmp);
359     }
360 
361     return;
362 }
363 
364 /*****************************/
365 
366 void
```

```
367 RoutingUnit::addPort(std::vector<PortDirection> &output_set, PortDirection dir,
        int my_x, int my_y, int num_rows, int num_cols, PortDirection inport_dirn)
368 {
369     if(std::find(output_set.begin(), output_set.end(), dir) != output_set.end())
370         return;
371
372     if (dir == "North" && inport_dirn != "North")
373     {
374         if (my_y != num_rows-1)
375             output_set.push_back(dir);
376     }
377
378     else if (dir == "South" && inport_dirn != "South")
379     {
380         if (my_y != 0)
381             output_set.push_back(dir);
382     }
383
384     else if (dir == "East" && inport_dirn != "East")
385     {
386         if (my_x != num_cols-1)
387             output_set.push_back(dir);
388     }
389
390     else if (dir == "West" && inport_dirn != "West")
391     {
392         if (my_x != 0)
393             output_set.push_back(dir);
394     }
395
396     else
397         return;
398 }
399
400 // ====> West First Routing
401
402 std::vector<OutputUnit*>
403 RoutingUnit::routing_WF(RouteInfo route, int current, Router *router,
        PortDirection inport_dirn)
404 {
405     PortDirection outport_dirn = "Unknown";
406
407     // Number of rows and number of columns
408     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
409     int num_cols = m_router->get_net_ptr()->getNumCols();
410     assert(num_rows > 0 && num_cols > 0);
411
412     // Source position
413     int my_id = current;
414     int my_x = my_id % num_cols;
415     int my_y = my_id / num_cols;
416
417     // Destination position
418     int dest_id = route.dest_router;
419     int dest_x = dest_id % num_cols;
420     int dest_y = dest_id / num_cols;
421
422     // Number of hops between source and destination in x and y direction
423     int x_hops = dest_x - my_x;
424     int y_hops = dest_y - my_y;
425
426     // already checked that in outportCompute() function
427     assert(!(x_hops == 0 && y_hops == 0));
428
429     // Possible output
430     std::vector<PortDirection> output_set;
431
432     if (x_hops < 0) {
433         output_set.push_back("West");
434     }
435     else if (x_hops > 0 && y_hops > 0) {
436         output_set.push_back("East");
437         output_set.push_back("North");
438     }
439     else if (x_hops > 0 && y_hops < 0) {
440         output_set.push_back("East");
441         output_set.push_back("South");
442     }
443     else if (x_hops > 0 && y_hops == 0) {
444          output_set.push_back("East");
```

```
445     }
446     else if (x_hops == 0 && y_hops > 0) {
447         output_set.push_back("North");
448     }
449     else if (x_hops == 0 && y_hops < 0) {
450         output_set.push_back("South");
451     }
452     else
453         assert(0);
454 
455 
456     std::vector<OutputUnit*> possible_output_set;
457 
458     for (auto elem : output_set)
459         possible_output_set.push_back(router->get_map_direction_out(elem));
460 
461     return possible_output_set;
462 }
463 
464 std::vector<std::vector<OutputUnit*>>
465 RoutingUnit::routing_NMWF(RouteInfo route, int current, Router *router,
      PortDirection inport_dirn)
466 {
467     PortDirection outport_dirn = "Unknown";
468 
469     // Number of rows and number of columns
470     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
471     int num_cols = m_router->get_net_ptr()->getNumCols();
472     assert(num_rows > 0 && num_cols > 0);
473 
474     // Source position
475     int my_id = current;
476     int my_x = my_id % num_cols;
477     int my_y = my_id / num_cols;
478 
479     // Destination position
480     int dest_id = route.dest_router;
481     int dest_x = dest_id % num_cols;
482     int dest_y = dest_id / num_cols;
483 
484     // Number of hops between source and destination in x and y direction
485     int x_hops = dest_x - my_x;
486     int y_hops = dest_y - my_y;
487 
488     // already checked that in outportCompute() function
489     assert(!(x_hops == 0 && y_hops == 0));
490 
491     // Possible output
492     std::vector<PortDirection> output_set0;
493     std::vector<PortDirection> output_set1;
494     std::vector<PortDirection> output_set2;
495 
496     if (x_hops == 0)
497     {
498 
499         if (y_hops > 0)
500         {
501             addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
      inport_dirn);
502 
503             if (inport_dirn == "East" || inport_dirn == "Local")
504                 addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
       inport_dirn);
505         }
506 
507         else
508         {
509             addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
      inport_dirn);
510 
511             if (inport_dirn == "East" || inport_dirn == "Local")
512                 addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
       inport_dirn);
513         }
514     }
515 
516     else if (x_hops != 0)
517     {
518         if (x_hops > 0)
519         {
```

```
520             addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
    inport_dirn);
521
522             if (y_hops > 0)
523             {
524                 addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
    inport_dirn);
525                 addPort(output_set2, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
526
527                 if (inport_dirn == "East" || inport_dirn == "Local")
528                     addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
529             }
530
531             else if (y_hops < 0)
532             {
533                 addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
534                 addPort(output_set2, "North", my_x, my_y, num_rows, num_cols,
    inport_dirn);
535
536                 if (inport_dirn == "East" || inport_dirn == "Local")
537                     addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
538             }
539
540             else
541             {
542                 addPort(output_set1, "North", my_x, my_y, num_rows, num_cols,
    inport_dirn);
543                 addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
544             }
545         }
546
547         else
548         {
549             if (inport_dirn == "East" || inport_dirn == "Local")
550                 addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
551         }
552     }
553
554     else
555         assert(0);
556
557     std::vector<std::vector<OutputUnit*>> possible_output_set;
558     std::vector<OutputUnit*> tmp0;
559     std::vector<OutputUnit*> tmp1;
560     std::vector<OutputUnit*> tmp2;
561     std::vector<OutputUnit*> tmp3;
562
563     for (auto elem : output_set0)
564     {
565         tmp0.push_back(router->get_map_direction_out(elem));
566         tmp1.push_back(router->get_map_direction_out(elem));
567     }
568
569     for (auto elem : output_set1)
570     {
571         tmp0.push_back(router->get_map_direction_out(elem));
572         tmp2.push_back(router->get_map_direction_out(elem));
573     }
574
575     for (auto elem : output_set2)
576     {
577         tmp0.push_back(router->get_map_direction_out(elem));
578         tmp3.push_back(router->get_map_direction_out(elem));
579     }
580
581     possible_output_set.push_back(tmp0);
582     possible_output_set.push_back(tmp1);
583     possible_output_set.push_back(tmp2);
584     possible_output_set.push_back(tmp3);
585
586     return possible_output_set;
587 }
588
589 void
```

```
590 RoutingUnit::outportComputeCustomWF(RouteInfo route, int current, PortDirection
        inport_dirn, Route &mroute, std::vector<Route> &setRoute)
591 {
592     // OE routing
593     Router* router = m_router->get_net_ptr()->getRouter(current);
594     std::vector<OutputUnit*> possible_output_set = routing_WF(route, current,
        router, inport_dirn);
595
596     for (auto port : possible_output_set)
597     {
598         mroute.push_back(port);
599         current = port->get_dst_router();
600
601         if (current == route.dest_router)
602         {
603             setRoute.push_back(mroute);
604             mroute.pop_back();
605         }
606
607         else
608         {
609             inport_dirn = map_out_in[port->get_direction()];
610             outportComputeCustomWF(route, current, inport_dirn, mroute, setRoute
        );
611         }
612     }
613     mroute.pop_back();
614     return;
615 }
616
617 /*****************************/
618
619 // ====> North Last Routing
620
621 std::vector<OutputUnit*>
622 RoutingUnit::routing_NL(RouteInfo route, int current, Router *router,
        PortDirection inport_dirn)
623 {
624     PortDirection outport_dirn = "Unknown";
625
626     // Number of rows and number of columns
627     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
628     int num_cols = m_router->get_net_ptr()->getNumCols();
629     assert(num_rows > 0 && num_cols > 0);
630
631     // Source position
632     int my_id = current;
633     int my_x = my_id % num_cols;
634     int my_y = my_id / num_cols;
635
636     // Destination position
637     int dest_id = route.dest_router;
638     int dest_x = dest_id % num_cols;
639     int dest_y = dest_id / num_cols;
640
641     // Number of hops between source and destination in x and y direction
642     int x_hops = dest_x - my_x;
643     int y_hops = dest_y - my_y;
644
645     // Possible output
646     std::vector<PortDirection> output_set;
647
648     // already checked that in outportCompute() function
649     assert(!(x_hops == 0 && y_hops == 0));
650
651     if (x_hops > 0 && y_hops > 0) {
652         // Move to east
653         output_set.push_back("East");
654     }
655     else if (x_hops > 0 && y_hops < 0) {
656         output_set.push_back("East"); // Could be East or South
657         output_set.push_back("South");
658     }
659     else if (x_hops > 0 && y_hops == 0) {
660         output_set.push_back("East");
661     }
662
663     else if (x_hops < 0 && y_hops > 0) {
664         // Move to east
665         output_set.push_back("West");
```

```
666     }
667     else if (x_hops < 0 && y_hops < 0) {
668         output_set.push_back("West"); // Could be West or South
669         output_set.push_back("South");
670     }
671     else if (x_hops < 0 && y_hops == 0) {
672         output_set.push_back("West");
673     }
674
675     else if (x_hops == 0 && y_hops > 0) {
676         output_set.push_back("North");
677     }
678     else if (x_hops == 0 && y_hops < 0) {
679         output_set.push_back("South");
680     }
681     else {
682         // x_hops == 0 and y_hops == 0
683         // this is not possible
684         // already checked that in outportCompute() function
685         assert(0);
686     }
687
688     std::vector<OutputUnit*> possible_output_set;
689     for (auto elem : output_set)
690     {
691         possible_output_set.push_back(router->get_map_direction_out(elem));
692     }
693
694     return possible_output_set;
695 }
696
697
698 std::vector<std::vector<OutputUnit*>>
699 RoutingUnit::routing_NMNL(RouteInfo route, int current, Router *router,
        PortDirection inport_dirn)
700 {
701     PortDirection outport_dirn = "Unknown";
702
703     // Number of rows and number of columns
704     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
705     int num_cols = m_router->get_net_ptr()->getNumCols();
706     assert(num_rows > 0 && num_cols > 0);
707
708     // Source position
709     int my_id = current;
710     int my_x = my_id % num_cols;
711     int my_y = my_id / num_cols;
712
713     // Destination position
714     int dest_id = route.dest_router;
715     int dest_x = dest_id % num_cols;
716     int dest_y = dest_id / num_cols;
717
718     // Number of hops between source and destination in x and y direction
719     int x_hops = dest_x - my_x;
720     int y_hops = dest_y - my_y;
721
722     // already checked that in outportCompute() function
723     assert(!(x_hops == 0 && y_hops == 0));
724
725     // Possible output
726     std::vector<PortDirection> output_set0;
727     std::vector<PortDirection> output_set1;
728     std::vector<PortDirection> output_set2;
729
730     if (x_hops == 0) {
731
732         if (y_hops > 0)
733         {
734             addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
        inport_dirn);
735         }
736
737         else
738         {
739             addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
        inport_dirn);
740             addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
        inport_dirn);
```

```
741                 addPort(output_set1, "East", my_x, my_y, num_rows, num_cols,
    inport_dirn);
742             }
743         }
744
745         else if (x_hops != 0)
746         {
747             if (x_hops > 0)
748             {
749                 addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
    inport_dirn);
750
751                 if (y_hops < 0)
752                 {
753                     addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
754                     addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
    inport_dirn);
755                 }
756
757                 else
758                     addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
759             }
760
761             else
762             {
763                 addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
    inport_dirn);
764
765                 if (y_hops < 0)
766                 {
767                     addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
768                     addPort(output_set2, "East", my_x, my_y, num_rows, num_cols,
    inport_dirn);
769                 }
770
771                 else
772                     addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
773             }
774         }
775
776         else
777             assert(0);
778
779         std::vector<std::vector<OutputUnit*>> possible_output_set;
780         std::vector<OutputUnit*> tmp0;
781         std::vector<OutputUnit*> tmp1;
782         std::vector<OutputUnit*> tmp2;
783         std::vector<OutputUnit*> tmp3;
784
785         for (auto elem : output_set0)
786         {
787             tmp0.push_back(router->get_map_direction_out(elem));
788             tmp1.push_back(router->get_map_direction_out(elem));
789         }
790
791         for (auto elem : output_set1)
792         {
793             tmp0.push_back(router->get_map_direction_out(elem));
794             tmp2.push_back(router->get_map_direction_out(elem));
795         }
796
797         for (auto elem : output_set2)
798         {
799             tmp0.push_back(router->get_map_direction_out(elem));
800             tmp3.push_back(router->get_map_direction_out(elem));
801         }
802
803         possible_output_set.push_back(tmp0);
804         possible_output_set.push_back(tmp1);
805         possible_output_set.push_back(tmp2);
806         possible_output_set.push_back(tmp3);
807
808         return possible_output_set;
809 }
810
811
```

```
812 void
813 RoutingUnit::outportComputeCustomNL(RouteInfo route, int current, PortDirection
        inport_dirn, Route &mroute, std::vector<Route> &setRoute)
814 {
815     // OE routing
816     Router* router = m_router->get_net_ptr()->getRouter(current);
817     std::vector<OutputUnit*> possible_output_set = routing_NL(route, current,
        router, inport_dirn);
818
819     for (auto port : possible_output_set)
820     {
821         mroute.push_back(port);
822         current = port->get_dst_router();
823
824         if (current == route.dest_router)
825         {
826             setRoute.push_back(mroute);
827             mroute.pop_back();
828         }
829
830         else
831         {
832             inport_dirn = map_out_in[port->get_direction()];
833             outportComputeCustomNL(route, current, inport_dirn, mroute, setRoute
        );
834         }
835     }
836     mroute.pop_back();
837     return;
838 }
839
840 /*****************************/
841
842 // ====> Negative First Routing
843
844 std::vector<OutputUnit*>
845 RoutingUnit::routing_NF(RouteInfo route, int current, Router *router,
        PortDirection inport_dirn)
846 {
847     PortDirection outport_dirn = "Unknown";
848
849     // Number of rows and number of columns
850     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
851     int num_cols = m_router->get_net_ptr()->getNumCols();
852     assert(num_rows > 0 && num_cols > 0);
853
854     // Source position
855     int my_id = current;
856     int my_x = my_id % num_cols;
857     int my_y = my_id / num_cols;
858
859     // Destination position
860     int dest_id = route.dest_router;
861     int dest_x = dest_id % num_cols;
862     int dest_y = dest_id / num_cols;
863
864     // Number of hops between source and destination in x and y direction
865     int x_hops = dest_x - my_x;
866     int y_hops = dest_y - my_y;
867
868     // Possible output
869     std::vector<PortDirection> output_set;
870
871     // already checked that in outportCompute() function
872     assert(!(x_hops == 0 && y_hops == 0));
873
874     if (x_hops < 0 && y_hops < 0) {
875         // Negative first
876         output_set.push_back("West"); // Could be West or South
877         output_set.push_back("South");
878     }
879     else if (x_hops < 0 && y_hops > 0) {
880         // Negative first
881         output_set.push_back("West");
882     }
883     else if (x_hops < 0 && y_hops == 0) {
884         // Negative first
885         output_set.push_back("West");
886     }
887     else if (x_hops > 0 && y_hops > 0) {
```

```
888             output_set.push_back("East"); // Could be East or North
889             output_set.push_back("North");
890         }
891         else if (x_hops > 0 && y_hops < 0) {
892             // Negative first
893             output_set.push_back("South");
894         }
895         else if (x_hops > 0 && y_hops == 0) {
896             output_set.push_back("East");
897         }
898
899         else if (x_hops == 0 && y_hops > 0) {
900             output_set.push_back("North");
901         }
902         else if (x_hops == 0 && y_hops < 0) {
903             output_set.push_back("South");
904         }
905         else {
906             // x_hops == 0 and y_hops == 0
907             // this is not possible
908             // already checked that in outportCompute() function
909             assert(0);
910         }
911
912         std::vector<OutputUnit*> possible_output_set;
913         for (auto elem : output_set)
914         {
915             possible_output_set.push_back(router->get_map_direction_out(elem));
916         }
917
918         return possible_output_set;
919     }
920
921
922     std::vector<std::vector<OutputUnit*>>
923     RoutingUnit::routing_NMNF(RouteInfo route, int current, Router *router,
          PortDirection inport_dirn)
924     {
925         PortDirection outport_dirn = "Unknown";
926
927         // Number of rows and number of columns
928         int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
929         int num_cols = m_router->get_net_ptr()->getNumCols();
930         assert(num_rows > 0 && num_cols > 0);
931
932         // Source position
933         int my_id = current;
934         int my_x = my_id % num_cols;
935         int my_y = my_id / num_cols;
936
937         // Destination position
938         int dest_id = route.dest_router;
939         int dest_x = dest_id % num_cols;
940         int dest_y = dest_id / num_cols;
941
942         // Number of hops between source and destination in x and y direction
943         int x_hops = dest_x - my_x;
944         int y_hops = dest_y - my_y;
945
946         // Possible output
947         std::vector<PortDirection> output_set0;
948         std::vector<PortDirection> output_set1;
949         std::vector<PortDirection> output_set2;
950
951         // already checked that in outportCompute() function
952         assert(!(x_hops == 0 && y_hops == 0));
953
954         if (x_hops != 0 && y_hops != 0)
955         {
956             if (x_hops > 0 && y_hops > 0)
957             {
958                 addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
          inport_dirn);
959                 addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
          inport_dirn);
960
961                 if (inport_dirn == "East" || inport_dirn == "North" || inport_dirn
          == "Local")
962                 {
```

```
963                 addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
    inport_dirn);
964                 addPort(output_set2, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
965             }
966         }
967
968         else
969         {
970             if (inport_dirn == "East" || inport_dirn == "North" || inport_dirn
    == "Local")
971             {
972                 addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
    inport_dirn);
973                 addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
974             }
975         }
976     }
977
978     else if (y_hops == 0)
979     {
980         if (x_hops > 0)
981         {
982             addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
    inport_dirn);
983
984             if (inport_dirn == "East" || inport_dirn == "North" || inport_dirn
    == "Local")
985             {
986                 if (my_y != 0)
987                     addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
988
989                 addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
990             }
991
992         }
993
994         if (x_hops < 0)
995         {
996             if (inport_dirn == "East" || inport_dirn == "North" || inport_dirn
    == "Local")
997             {
998                 addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
    inport_dirn);
999                 addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
    inport_dirn);
1000            }
1001        }
1002    }
1003
1004
1005    else if (x_hops == 0)
1006    {
1007        if (y_hops > 0)
1008        {
1009            addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
    inport_dirn);
1010
1011            if (my_x != 0)
1012            {
1013                if (inport_dirn == "East" || inport_dirn == "North" ||
    inport_dirn == "Local")
1014                {
1015                    addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1016                    addPort(output_set2, "South", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1017                }
1018            }
1019        }
1020
1021        if (y_hops < 0)
1022        {
1023            if (inport_dirn == "East" || inport_dirn == "North" || inport_dirn
    == "Local")
1024            {
```

```
1025                    addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1026                    addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1027                }
1028            }
1029        }
1030
1031        else
1032            assert(0);
1033
1034        std::vector<std::vector<OutputUnit*>> possible_output_set;
1035        std::vector<OutputUnit*> tmp0;
1036        std::vector<OutputUnit*> tmp1;
1037        std::vector<OutputUnit*> tmp2;
1038        std::vector<OutputUnit*> tmp3;
1039
1040        for (auto elem : output_set0)
1041        {
1042            tmp0.push_back(router->get_map_direction_out(elem));
1043            tmp1.push_back(router->get_map_direction_out(elem));
1044        }
1045
1046        for (auto elem : output_set1)
1047        {
1048            tmp0.push_back(router->get_map_direction_out(elem));
1049            tmp2.push_back(router->get_map_direction_out(elem));
1050        }
1051
1052        for (auto elem : output_set2)
1053        {
1054            tmp0.push_back(router->get_map_direction_out(elem));
1055            tmp3.push_back(router->get_map_direction_out(elem));
1056        }
1057
1058        possible_output_set.push_back(tmp0);
1059        possible_output_set.push_back(tmp1);
1060        possible_output_set.push_back(tmp2);
1061        possible_output_set.push_back(tmp3);
1062        return possible_output_set;
1063    }
1064
1065
1066    void
1067    RoutingUnit::outportComputeCustomNF(RouteInfo route, int current, PortDirection
     inport_dirn, Route &mroute, std::vector<Route> &setRoute)
1068    {
1069        // OE routing
1070        Router* router = m_router->get_net_ptr()->getRouter(current);
1071        std::vector<OutputUnit*> possible_output_set = routing_NF(route, current,
     router, inport_dirn);
1072
1073        for (auto port : possible_output_set)
1074        {
1075            mroute.push_back(port);
1076            current = port->get_dst_router();
1077
1078            if (current == route.dest_router)
1079            {
1080                setRoute.push_back(mroute);
1081                mroute.pop_back();
1082            }
1083
1084            else
1085            {
1086                inport_dirn = map_out_in[port->get_direction()];
1087                outportComputeCustomNF(route, current, inport_dirn, mroute, setRoute
     );
1088            }
1089        }
1090        mroute.pop_back();
1091        return;
1092    }
1093
1094    /************************/
1095
1096    // ====> Odd-Even routing
1097
1098    std::vector<OutputUnit*>
```

```
1099 RoutingUnit::routing_OE(RouteInfo route, int current, Router *router,
         PortDirection inport_dirn)
1100 {
1101     PortDirection outport_dirn = "Unknown";
1102
1103     // Number of rows and number of columns
1104     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
1105     int num_cols = m_router->get_net_ptr()->getNumCols();
1106     assert(num_rows > 0 && num_cols > 0);
1107
1108     // Current position
1109     int my_id = current;
1110     int my_x = my_id % num_cols;
1111     int my_y = my_id / num_cols;
1112
1113     // Destination position
1114     int dest_id = route.dest_router;
1115     int dest_x = dest_id % num_cols;
1116     int dest_y = dest_id / num_cols;
1117
1118     // Source position
1119     //int src_id = src;
1120     int src_id = route.src_router;
1121     int src_x = src_id % num_cols;
1122     //int src_y = src_id / num_cols;
1123
1124     // Number of hops between current and destination in x and y direction
1125     int x_hops = dest_x - my_x;
1126     int y_hops = dest_y - my_y;
1127
1128     // already checked that in outportCompute() function
1129     assert(!(x_hops == 0 && y_hops == 0));
1130
1131     // Possible output
1132     std::vector<PortDirection> output_set;
1133
1134     // Current switch is in the right column
1135     if (x_hops == 0)
1136     {
1137         if (y_hops > 0)
1138             output_set.push_back("North");
1139
1140         else if (y_hops < 0)
1141             output_set.push_back("South");
1142
1143     }
1144
1145     else
1146     {
1147         if (x_hops > 0)
1148         {
1149             if (y_hops == 0)
1150                 output_set.push_back("East");
1151
1152             else
1153             {
1154                 if (my_x % 2 != 0 || src_x == my_x)
1155                 {
1156                     if (y_hops > 0)
1157                         output_set.push_back("North");
1158                     else
1159                         output_set.push_back("South");
1160                 }
1161
1162                 if (dest_x % 2 != 0 || x_hops != 1)
1163                     output_set.push_back("East");
1164             }
1165         }
1166
1167         else
1168         {
1169             output_set.push_back("West");
1170             if (my_x % 2 == 0 && y_hops != 0)
1171             {
1172                 if (y_hops > 0)
1173                     output_set.push_back("North");
1174                 else
1175                     output_set.push_back("South");
1176             }
1177         }
```

```
1178     }
1179
1180     std::vector<OutputUnit*> possible_output_set;
1181     for (auto elem : output_set)
1182     {
1183         possible_output_set.push_back(router->get_map_direction_out(elem));
1184     }
1185
1186     return possible_output_set;
1187 }
1188
1189 std::vector<std::vector<OutputUnit*>>
1190 RoutingUnit::routing_NMOE(RouteInfo route, int current, Router *router,
          PortDirection inport_dirn)
1191 {
1192     PortDirection outport_dirn = "Unknown";
1193
1194     // Number of rows and number of columns
1195     int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
1196     int num_cols = m_router->get_net_ptr()->getNumCols();
1197     assert(num_rows > 0 && num_cols > 0);
1198
1199     // Current position
1200     int my_id = current;
1201     int my_x = my_id % num_cols;
1202     int my_y = my_id / num_cols;
1203
1204     // Parity of the current column
1205     //bool even = (my_x % 2 == 0);
1206
1207     // Destination position
1208     int dest_id = route.dest_router;
1209     int dest_x = dest_id % num_cols;
1210     int dest_y = dest_id / num_cols;
1211
1212
1213     // Number of hops between current and destination in x and y direction
1214     int x_hops = dest_x - my_x;
1215     int y_hops = dest_y - my_y;
1216
1217     // already checked that in outportCompute() function
1218     assert(!(x_hops == 0 && y_hops == 0));
1219
1220     // Possible output
1221     std::vector<PortDirection> output_set0;
1222     std::vector<PortDirection> output_set1;
1223     std::vector<PortDirection> output_set2;
1224
1225     // Same column
1226     if (x_hops == 0)
1227     {
1228         if (my_x % 2 != 0)
1229         {
1230             if (inport_dirn == "East")
1231                 addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
          inport_dirn);
1232
1233             if (y_hops < 0)
1234                 addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
          inport_dirn);
1235
1236             else
1237                 addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
          inport_dirn);
1238         }
1239
1240         else
1241         {
1242             addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
          inport_dirn);
1243
1244             if (y_hops < 0)
1245             {
1246                 addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
          inport_dirn);
1247
1248                 if (dest_x != 0)
1249                     addPort(output_set2, "North", my_x, my_y, num_rows, num_cols
          , inport_dirn);
1250             }
```

```
1251
1252                else
1253                {
1254                    addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1255
1256                    if (dest_x != 0)
1257                        addPort(output_set2, "South", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1258                }
1259            }
1260        }
1261
1262        // Same row
1263        else if (y_hops == 0)
1264        {
1265            if (my_x % 2 != 0)
1266            {
1267                if (x_hops > 0)
1268                {
1269                    addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1270
1271                    if (x_hops > 1)
1272                    {
1273                        addPort(output_set1, "North", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1274                        addPort(output_set1, "South", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1275                    }
1276
1277                    if (inport_dirn == "East")
1278                        addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1279                }
1280
1281                else
1282                {
1283                    addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1284                }
1285            }
1286
1287            else
1288            {
1289                if (x_hops > 0)
1290                {
1291                    addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1292                    addPort(output_set2, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1293
1294                    if (inport_dirn != "West")
1295                    {
1296                        addPort(output_set1, "North", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1297                        addPort(output_set1, "South", my_x, my_y, num_rows, num_cols
    , inport_dirn);
1298                    }
1299                }
1300
1301                else
1302                {
1303                    addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1304                    addPort(output_set1, "North", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1305                    addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1306                }
1307            }
1308        }
1309
1310        // North East
1311        else if (x_hops > 0 && y_hops > 0)
1312        {
1313            if (my_x % 2 != 0)
1314            {
1315                addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
     inport_dirn);
```

```
1316
1317                if (x_hops > 1)
1318                {
1319                    addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1320                    addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1321                }
1322
1323                if (inport_dirn == "East")
1324                    addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1325            }
1326
1327            else
1328            {
1329                addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1330                addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1331
1332                if (inport_dirn != "West")
1333                {
1334                    addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1335                    addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1336                }
1337            }
1338        }
1339
1340        // South East
1341        else if (x_hops > 0 && y_hops < 0)
1342        {
1343            if (my_x % 2 != 0)
1344            {
1345                addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1346
1347                if (x_hops > 1)
1348                {
1349                    addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1350                    addPort(output_set1, "North", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1351                }
1352
1353                if (inport_dirn == "East")
1354                    addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1355            }
1356
1357            else
1358            {
1359                addPort(output_set0, "East", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1360                addPort(output_set1, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1361
1362                if (inport_dirn != "West")
1363                {
1364                    addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1365                    addPort(output_set1, "North", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1366                }
1367            }
1368        }
1369
1370        // North West
1371        else if (x_hops < 0 && y_hops > 0)
1372        {
1373            if (my_x % 2 != 0)
1374                addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
1375            else
1376            {
1377                addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
      inport_dirn);
```

```
1378                addPort(output_set0, "North", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1379                addPort(output_set1, "South", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1380            }
1381        }
1382
1383        // South West
1384        else if (x_hops < 0 && y_hops < 0)
1385        {
1386            if (my_x % 2 != 0)
1387                addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1388            else
1389            {
1390                addPort(output_set0, "West", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1391                addPort(output_set0, "South", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1392                addPort(output_set1, "North", my_x, my_y, num_rows, num_cols,
     inport_dirn);
1393            }
1394        }
1395
1396        else
1397            assert(0);
1398
1399        std::vector<std::vector<OutputUnit*>> possible_output_set;
1400        std::vector<OutputUnit*> tmp0;
1401        std::vector<OutputUnit*> tmp1;
1402        std::vector<OutputUnit*> tmp2;
1403        std::vector<OutputUnit*> tmp3;
1404
1405        for (auto elem : output_set0)
1406        {
1407            tmp0.push_back(router->get_map_direction_out(elem));
1408            tmp1.push_back(router->get_map_direction_out(elem));
1409        }
1410
1411        for (auto elem : output_set1)
1412        {
1413            tmp0.push_back(router->get_map_direction_out(elem));
1414            tmp2.push_back(router->get_map_direction_out(elem));
1415        }
1416
1417        for (auto elem : output_set2)
1418        {
1419            tmp0.push_back(router->get_map_direction_out(elem));
1420            tmp3.push_back(router->get_map_direction_out(elem));
1421        }
1422
1423        possible_output_set.push_back(tmp0);
1424        possible_output_set.push_back(tmp1);
1425        possible_output_set.push_back(tmp2);
1426        possible_output_set.push_back(tmp3);
1427
1428        return possible_output_set;
1429    }
1430
1431    void
1432    RoutingUnit::outportComputeCustomOE(RouteInfo route, int current, PortDirection
     inport_dirn, Route &mroute, std::vector<Route> &setRoute)
1433    {
1434        // OE routing
1435        Router* router = m_router->get_net_ptr()->getRouter(current);
1436        std::vector<OutputUnit*> possible_output_set = routing_OE(route, current,
     router, inport_dirn);
1437
1438        for (auto port : possible_output_set)
1439        {
1440            mroute.push_back(port);
1441            current = port->get_dst_router();
1442
1443            if (current == route.dest_router)
1444            {
1445                setRoute.push_back(mroute);
1446                mroute.pop_back();
1447            }
1448
1449            else
```

```
1450            {
1451                inport_dirn = map_out_in[port->get_direction()];
1452                outportComputeCustomOE(route, current, inport_dirn, mroute, setRoute
        );
1453            }
1454        }
1455        mroute.pop_back();
1456        return;
1457    }
1458
1459    void
1460    RoutingUnit::outportComputeCustomNMOE(RouteInfo route, int current,
         PortDirection inport_dirn, Route &mroute, std::vector<Route> &setRoute)
1461    {
1462        // OE routing
1463        Router* router = m_router->get_net_ptr()->getRouter(current);
1464        std::vector<OutputUnit*> possible_output_set = routing_NMOE(route, current,
         router, inport_dirn)[0];
1465
1466        for (auto port : possible_output_set)
1467        {
1468            mroute.push_back(port);
1469            current = port->get_dst_router();
1470
1471            if (current == route.dest_router)
1472            {
1473                setRoute.push_back(mroute);
1474                mroute.pop_back();
1475            }
1476
1477            else
1478            {
1479                inport_dirn = map_out_in[port->get_direction()];
1480                outportComputeCustomNMOE(route, current, inport_dirn, mroute,
         setRoute);
1481            }
1482        }
1483        mroute.pop_back();
1484        return;
1485    }
1486
1487
1488    //************************
1489
1490    std::vector<OutputUnit*>
1491    RoutingUnit::routing_tmp(RouteInfo route, int new_src, int cnt, PortDirection
         inport_dirn)
1492    {
1493        RoutingAlgorithm routing_algorithm = (RoutingAlgorithm) m_router->
         get_net_ptr()->getRoutingAlgorithm();
1494
1495        if (routing_algorithm == WEST_FIRST_NOP || routing_algorithm ==
         WEST_FIRST_MEAN)
1496            return routing_WF(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn);
1497
1498        else if (routing_algorithm == NORTH_LAST_NOP || routing_algorithm ==
         NORTH_LAST_MEAN)
1499            return routing_NL(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn);
1500
1501        else if (routing_algorithm == NEGATIVE_FIRST_NOP || routing_algorithm ==
         NEGATIVE_FIRST_MEAN)
1502            return routing_NF(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn);
1503
1504        else if (routing_algorithm == ODD_EVEN_NOP || routing_algorithm ==
         ODD_EVEN_MEAN)
1505            return routing_OE(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn);
1506
1507        else if (routing_algorithm == NM_WEST_FIRST_NOP || routing_algorithm ==
         NM_WEST_FIRST_MEAN)
1508            return routing_NMWF(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn)[0];
1509
1510        else if (routing_algorithm == NM_NORTH_LAST_NOP || routing_algorithm ==
         NM_NORTH_LAST_MEAN)
1511            return routing_NMNL(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn)[0];
```

```
1512
1513     else if (routing_algorithm == NM_NEGATIVE_FIRST_NOP || routing_algorithm ==
         NM_NEGATIVE_FIRST_MEAN)
1514         return routing_NMNF(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn)[0];
1515
1516     else if (routing_algorithm == NM_ODD_EVEN_NOP || routing_algorithm ==
         NM_ODD_EVEN_MEAN)
1517         return routing_NMOE(route, new_src, m_router->get_net_ptr()->getRouter(
         new_src), inport_dirn)[0];
1518
1519     else
1520         assert(0);
1521 }
1522
1523 /******************/
1524
1525 // ======> Mean Selection
1526
1527 void
1528 RoutingUnit::recursive_routing(RouteInfo route, int current, int dst, std::
         vector<int> &buffer, OutputUnit* outc, int* cnt, int* cnt_ud,
1529                                std::map<OutputUnit*, double> &scores,
         PortDirection inport_dirn, std::vector<int> &buf_r)
1530 {
1531
1532     (*cnt_ud)++;
1533     std::vector<OutputUnit*> admissible_outc = routing_tmp(route, current, *
         cnt_ud, inport_dirn);
1534
1535     if(std::find(buf_r.begin(), buf_r.end(), current) == buf_r.end()) {
1536         (*cnt)++;
1537         buf_r.push_back(current);
1538
1539         // Score
1540         int nb_out = m_router->get_net_ptr()->getRouter(current)->
         get_num_outports();
1541         for (int i = 2; i < nb_out; i++)
1542         {
1543             std::vector<OutputUnit*> tmp = m_router->get_net_ptr()->getRouter(
         current)->get_outputUnit_ref();
1544
1545             if (tmp[i]->get_dst_router() != m_router->get_net_ptr()->
         getNumRouters())
1546             {
1547                 int tmp_router = tmp[i]->get_dst_router();
1548                 std::vector<InputUnit*> inport_tmp = m_router->get_net_ptr()->
         getRouter(tmp_router)->get_inputUnit_ref();
1549                 for (int j = 0; j < inport_tmp.size(); j++)
1550                 {
1551                     if (inport_tmp[j]->get_direction() == "Local" || inport_tmp[
         j]->get_direction() == "to_node_dst")
1552                         continue;
1553                     double sc = (double)inport_tmp[j]->get_free_slots_in(route.
         vnet);
1554                     scores[outc] += sc / (double)(inport_tmp.size()-3) / (double
         )(nb_out-3);
1555                 }
1556             }
1557         }
1558
1559     }
1560
1561     for (auto neigh_c : admissible_outc)
1562     {
1563         int neigh = neigh_c->get_dst_router();
1564         if(std::find(buffer.begin(), buffer.end(), neigh_c->get_outlink_id()) !=
          buffer.end())
1565             continue;
1566
1567         buffer.push_back(neigh_c->get_outlink_id());
1568
1569         if (neigh == dst)
1570             continue;
1571
1572         recursive_routing(route, neigh, dst, buffer, outc, cnt, cnt_ud, scores,
         map_out_in[neigh_c->get_direction()], buf_r);
1573     }
1574
1575     (*cnt_ud)--;
```

```
1576
1577        return;
1578    }
1579
1580    type_outScore
1581    RoutingUnit::meanSelection(RouteInfo route, std::vector<OutputUnit*> &
            free_output_set)
1582    {
1583        std::vector<int> buffer;
1584        std::vector<int> buf_r;
1585        std::map<OutputUnit*, double> scores;
1586        int dst = route.dest_router;
1587
1588        for (auto outc : free_output_set) {
1589            int cnt = 1;
1590            int cnt_ud = 0;
1591            buffer = {outc->get_outlink_id()};
1592            buf_r = {outc->get_dst_router()};
1593
1594            // Score 3.0
1595            int nb_out = m_router->get_net_ptr()->getRouter(outc->get_dst_router())
        ->get_num_outports();
1596            for (int i = 2; i < nb_out; i++) {
1597
1598                std::vector<OutputUnit*> tmp = m_router->get_net_ptr()->getRouter(
        outc->get_dst_router())->get_outputUnit_ref();
1599                int tmp_router = tmp[i]->get_dst_router();
1600
1601                if (tmp_router != m_router->get_net_ptr()->getNumRouters()) {
1602
1603                    std::vector<InputUnit*> inport_tmp = m_router->get_net_ptr()->
        getRouter(tmp_router)->get_inputUnit_ref();
1604
1605                    for (int j = 0; j < inport_tmp.size(); j++)
1606                    {
1607                        if (inport_tmp[j]->get_direction() == "Local" || inport_tmp[
        j]->get_direction() == "to_node_dst")
1608                            continue;
1609
1610                        double sc = (double) inport_tmp[j]->get_free_slots_in(route.
        vnet);
1611                        scores[outc] += sc / (double)(inport_tmp.size()-3) / (double
        )(nb_out-3);
1612                    }
1613                }
1614            }
1615
1616            recursive_routing(route, outc->get_dst_router(), dst, buffer, outc, &cnt
        , &cnt_ud, scores, map_out_in[outc->get_direction()], buf_r);
1617            scores[outc] /= (double) cnt ;
1618        }
1619
1620
1621        // Select MIN scores
1622        double min_score = INFINITE_;
1623        std::vector<OutputUnit*> output_link_candidates;
1624        int num_candidates = 0;
1625
1626        // Check the MIN score
1627        for (auto out : free_output_set)
1628        {
1629            if (scores[out] < min_score)
1630                min_score = scores[out];
1631        }
1632
1633        // Check the outport with the MIN score
1634        for (auto out : free_output_set)
1635        {
1636            if (scores[out] == min_score)
1637            {
1638                output_link_candidates.push_back(out);
1639                num_candidates++;
1640            }
1641        }
1642
1643        // Select one of the canditate
1644        OutputUnit* outport = output_link_candidates[rand() % num_candidates];
1645
1646        // Return
1647        return std::make_pair(outport, min_score);
```

```
1648 }
1649
1650 type_outScore
1651 RoutingUnit::selection(RouteInfo route, std::vector<OutputUnit*> &
         possible_output_set, bool one)
1652 {
1653     // 1) No outport possible
1654     if (possible_output_set.size() == 0)
1655         return std::make_pair(nullptr, INFINITE_);
1656
1657     // If channel adjacent to the dest, take it
1658     for (auto out : possible_output_set)
1659     {
1660         if (out->get_dst_router() == route.dest_router)
1661             return std::make_pair(out, 0);
1662     }
1663
1664     // Check best path
1665     type_outScore outport_score = meanSelection(route, possible_output_set);
1666
1667     return outport_score;
1668 }
1669
1670 /*************************/
1671 Route
1672 RoutingUnit::selection_link_max(std::vector<Route> &setRoute, RouteInfo
         routeinfo)
1673 {
1674
1675     std::vector<double> scores;
1676
1677     for (int i = 0; i < setRoute.size(); i++)
1678     {
1679         Route route = setRoute[i];
1680         double score = 0;
1681
1682         for (int n = 0; n < route.size() - 1; n++)
1683         {
1684             double tmp_score = 0;
1685
1686             //*****************************************
1687             // Link Utilization
1688             InputUnit* in = m_router->get_net_ptr()->get_map_link_inport(route[n
         ]->get_nwk_link());
1689             tmp_score += in->get_state_rate_saved();
1690             //*****************************************
1691
1692
1693             // Most congested router
1694             if (tmp_score > score)
1695                 score = tmp_score;
1696
1697             // // Sum
1698             // score += tmp_score;
1699         }
1700
1701         //*****************************************
1702         // Hop count
1703         double hop_diff = (double) route.size() - (double) setRoute[0].size();
1704         double num = (double) m_router->get_net_ptr()->getNumRouters();
1705         double penalty = beta * hop_diff / num;
1706         score += penalty;
1707         //*****************************************
1708
1709         scores.push_back(score);
1710     }
1711
1712
1713     // Select MIN scores
1714     double min_score = INFINITE_;
1715     std::vector<Route> route_candidates;
1716     int num_candidates = 0;
1717
1718     // Check the MIN score
1719     for (auto score : scores)
1720     {
1721         if (score < min_score)
1722             min_score = score;
1723     }
1724
```

```
1725     // Check the route with the MIN score
1726     for (int i = 0; i < scores.size(); i++)
1727     {
1728         if (scores[i] == min_score)
1729         {
1730             route_candidates.push_back(setRoute[i]);
1731             num_candidates++;
1732         }
1733     }
1734 
1735     // Select one of the canditate
1736     Route final_route = route_candidates[rand() % num_candidates];
1737     return final_route;
1738 }
1739 
1740 Route
1741 RoutingUnit::selection_link_sum(std::vector<Route> &setRoute, RouteInfo
         routeinfo)
1742 {
1743 
1744     std::vector<double> scores;
1745 
1746     for (int i = 0; i < setRoute.size(); i++)
1747     {
1748         Route route = setRoute[i];
1749         double score = 0;
1750 
1751         for (int n = 0; n < route.size() - 1; n++)
1752         {
1753             double tmp_score = 0;
1754 
1755             //**********************************
1756             // Link Utilization
1757             InputUnit* in = m_router->get_net_ptr()->get_map_link_inport(route[n
         ]->get_nwk_link());
1758             tmp_score += in->get_state_rate_saved();
1759             //**********************************
1760 
1761             score += tmp_score;
1762         }
1763 
1764 
1765         scores.push_back(score);
1766     }
1767 
1768 
1769 
1770 
1771     // Select MIN scores
1772     double min_score = INFINITE_;
1773     std::vector<Route> route_candidates;
1774     int num_candidates = 0;
1775 
1776     // Check the MIN score
1777     for (auto score : scores)
1778     {
1779         if (score < min_score)
1780             min_score = score;
1781     }
1782 
1783     // Check the route with the MIN score
1784     for (int i = 0; i < scores.size(); i++)
1785     {
1786         if (scores[i] == min_score)
1787         {
1788             route_candidates.push_back(setRoute[i]);
1789             num_candidates++;
1790         }
1791     }
1792 
1793     // Select one of the canditate
1794     Route final_route = route_candidates[rand() % num_candidates];
1795     return final_route;
1796 }
1797 
1798 Route
1799 RoutingUnit::selection_router_max(std::vector<Route> &setRoute, RouteInfo
         routeinfo)
1800 {
1801 
```

```
1802      std::vector<double> scores;
1803
1804      for (int i = 0; i < setRoute.size(); i++)
1805      {
1806          Route route = setRoute[i];
1807          double score = 0;
1808          double link_load = 0;
1809          double router_load = 0;
1810          //double penality = 0;
1811
1812          for (int n = 0; n < route.size() - 1; n++)
1813          {
1814              //****************************************
1815              // Link Utilization
1816              InputUnit* in = m_router->get_net_ptr()->get_map_link_inport(route[n
      ]->get_nwk_link());
1817              double tmp_link_load = in->get_state_rate_saved();
1818
1819              //****************************************
1820
1821              //****************************************
1822              // Traffic rate arriving on the Current router
1823              Router* router = route[n]->get_router();
1824              std::vector<InputUnit*> inputs = router->get_inputUnit_ref();
1825              double tmp_router_load = 0;
1826              for (int j = 0; j < inputs.size(); j++)
1827              {
1828                if (inputs[j]->get_direction() != route[n]->get_direction())
1829                    tmp_router_load += inputs[j]->get_state_rate_saved();
1830              }
1831
1832              router_load = router_load / (double) (inputs.size() - 1);
1833              //****************************************
1834              // Most congested link
1835              if (tmp_link_load > link_load)
1836                  link_load = tmp_link_load;
1837
1838              // Most congested router
1839              if (tmp_router_load > router_load)
1840                  router_load = tmp_router_load;
1841
1842          }
1843
1844          score = router_load + link_load;
1845          scores.push_back(score);
1846      }
1847
1848
1849
1850
1851      // Select MIN scores
1852      double min_score = INFINITE_;
1853      std::vector<Route> route_candidates;
1854      int num_candidates = 0;
1855
1856      // Check the MIN score
1857      for (auto score : scores)
1858      {
1859          if (score < min_score)
1860              min_score = score;
1861      }
1862
1863      // Check the route with the MIN score
1864      for (int i = 0; i < scores.size(); i++)
1865      {
1866          if (scores[i] == min_score)
1867          {
1868              route_candidates.push_back(setRoute[i]);
1869              num_candidates++;
1870          }
1871      }
1872
1873      // Select one of the canditate
1874      Route final_route = route_candidates[rand() % num_candidates];
1875      return final_route;
1876  }
1877
1878  Route
1879  RoutingUnit::selection_router_sum(std::vector<Route> &setRoute, RouteInfo
       routeinfo)
```

```
1880 {
1881
1882
1883     std::vector<double> scores;
1884     for (int i = 0; i < setRoute.size(); i++)
1885     {
1886         Route route = setRoute[i];
1887
1888         double score = 0;
1889
1890         for (int n = 0; n < route.size() - 1; n++)
1891         {
1892             double tmp_score = 0;
1893
1894             //**************************************
1895             // Link Utilization
1896             InputUnit* in = m_router->get_net_ptr()->get_map_link_inport(route[n
     ]->get_nwk_link());
1897             tmp_score += in->get_state_rate_saved();
1898             //**************************************
1899
1900             //**************************************
1901             // Traffic rate arriving on the Current router
1902             Router* router = route[n]->get_router();
1903             std::vector<InputUnit*> inputs = router->get_inputUnit_ref();
1904             double router_load = 0;
1905             for (int j = 0; j < inputs.size(); j++)
1906             {
1907               if (inputs[j]->get_direction() != route[n]->get_direction())
1908                   router_load += inputs[j]->get_state_rate_saved();
1909             }
1910
1911             router_load = gamma * router_load / (double) (inputs.size() - 1);
1912             tmp_score += router_load;
1913             //******************************************
1914
1915             // Sum
1916             score += tmp_score;
1917         }
1918
1919         score /= (double) route.size();
1920
1921
1922         scores.push_back(score);
1923     }
1924
1925
1926     // Select MIN scores
1927     double min_score = INFINITE_;
1928     std::vector<Route> route_candidates;
1929     int num_candidates = 0;
1930
1931     // Check the MIN score
1932     for (auto score : scores)
1933     {
1934         if (score < min_score)
1935             min_score = score;
1936     }
1937
1938     // Check the route with the MIN score
1939     for (int i = 0; i < scores.size(); i++)
1940     {
1941         if (scores[i] == min_score)
1942         {
1943             route_candidates.push_back(setRoute[i]);
1944             num_candidates++;
1945         }
1946     }
1947
1948     // Select one of the canditate
1949     Route final_route = route_candidates[rand() % num_candidates];
1950     return final_route;
1951 }
1952
1953 /*****************************/
1954 std::vector<OutputUnit*>
1955 RoutingUnit::routing_XY(RouteInfo route, int current, Router *router,
          PortDirection inport_dirn)
1956 {
1957     PortDirection outport_dirn = "Unknown";
```

```
1958
1959        int M5_VAR_USED num_rows = m_router->get_net_ptr()->getNumRows();
1960        int num_cols = m_router->get_net_ptr()->getNumCols();
1961        assert(num_rows > 0 && num_cols > 0);
1962
1963        //int my_id = m_router->get_id();
1964        int my_id = current;
1965        int my_x = my_id % num_cols;
1966        int my_y = my_id / num_cols;
1967
1968        int dest_id = route.dest_router;
1969        int dest_x = dest_id % num_cols;
1970        int dest_y = dest_id / num_cols;
1971
1972        int x_hops = abs(dest_x - my_x);
1973        int y_hops = abs(dest_y - my_y);
1974
1975        bool x_dirn = (dest_x >= my_x);
1976        bool y_dirn = (dest_y >= my_y);
1977
1978        // already checked that in outportCompute() function
1979        assert(!(x_hops == 0 && y_hops == 0));
1980
1981        // Possible output
1982        std::vector<PortDirection> output_set;
1983
1984
1985        if (x_hops > 0) {
1986            if (x_dirn) {
1987                assert(inport_dirn == "Local" || inport_dirn == "West");
1988                output_set.push_back("East");
1989            } else {
1990                assert(inport_dirn == "Local" || inport_dirn == "East");
1991                output_set.push_back("West");
1992            }
1993        } else if (y_hops > 0) {
1994            if (y_dirn) {
1995                // "Local" or "South" or "West" or "East"
1996                assert(inport_dirn != "North");
1997                output_set.push_back("North");
1998            } else {
1999                // "Local" or "North" or "West" or "East"
2000                assert(inport_dirn != "South");
2001                output_set.push_back("South");
2002            }
2003        } else {
2004
2005            assert(0);
2006        }
2007
2008        std::vector<OutputUnit*> possible_output_set;
2009        for (auto elem : output_set)
2010        {
2011            possible_output_set.push_back(router->get_map_direction_out(elem));
2012        }
2013        return possible_output_set;
2014    }
2015
2016    void
2017    RoutingUnit::outportComputeCustomXY(RouteInfo route, int current, PortDirection
        inport_dirn, Route &mroute, std::vector<Route> &setRoute)
2018    {
2019        // XY routing
2020        Router* router = m_router->get_net_ptr()->getRouter(current);
2021        std::vector<OutputUnit*> possible_output_set = routing_XY(route, current,
        router, inport_dirn);
2022
2023        for (auto port : possible_output_set)
2024        {
2025            mroute.push_back(port);
2026            current = port->get_dst_router();
2027
2028            if (current == route.dest_router)
2029            {
2030                setRoute.push_back(mroute);
2031                mroute.pop_back();
2032            }
2033
2034            else
2035            {
```

```
2036                inport_dirn = map_out_in[port->get_direction()];
2037                outportComputeCustomXY(route, current, inport_dirn, mroute, setRoute
        );
2038            }
2039        }
2040        mroute.pop_back();
2041
2042        return;
2043    }
```

Listing A.1: RoutingUnit.cc

```
1   #include "mem/ruby/network/garnet2.0/Router.hh"
2
3   #include "base/stl_helpers.hh"
4   #include "debug/RubyNetwork.hh"
5   #include "mem/ruby/network/garnet2.0/CreditLink.hh"
6   #include "mem/ruby/network/garnet2.0/CrossbarSwitch.hh"
7   #include "mem/ruby/network/garnet2.0/GarnetNetwork.hh"
8   #include "mem/ruby/network/garnet2.0/InputUnit.hh"
9   #include "mem/ruby/network/garnet2.0/NetworkLink.hh"
10  #include "mem/ruby/network/garnet2.0/OutputUnit.hh"
11  #include "mem/ruby/network/garnet2.0/RoutingUnit.hh"
12  #include "mem/ruby/network/garnet2.0/SwitchAllocator.hh"
13
14  using namespace std;
15  using m5::stl_helpers::deletePointers;
16
17  Router::Router(const Params *p)
18      : BasicRouter(p), Consumer(this)
19  {
20      m_latency = p->latency;
21      m_virtual_networks = p->virt_nets;
22      m_vc_per_vnet = p->vcs_per_vnet;
23      m_num_vcs = m_virtual_networks * m_vc_per_vnet;
24
25      m_routing_unit = new RoutingUnit(this);
26      m_sw_alloc = new SwitchAllocator(this);
27      m_switch = new CrossbarSwitch(this);
28
29      m_input_unit.clear();
30      m_output_unit.clear();
31
32      // // changes by Soultana Ellinidou-SDNoC
33      std::ifstream reader;
34      reader.open("/home/gaurav/gem5/var/tau.txt");
35      if (!reader)
36          assert(0);
37      reader >> tau;
38      reader.close();
39
40      std::ifstream reader2;
41      reader2.open("/home/gaurav/gem5/var/alpha.txt");
42      if (!reader2)
43          assert(0);
44      reader2 >> alpha;
45      reader2.close();
46
47      schedule_wakeup(Cycles(0));
48
49  }
50
51  Router::~Router()
52  {
53      deletePointers(m_input_unit);
54      deletePointers(m_output_unit);
55      delete m_routing_unit;
56      delete m_sw_alloc;
57      delete m_switch;
58  }
59
60  void
61  Router::init()
62  {
63      BasicRouter::init();
64
65      m_sw_alloc->init();
66      m_switch->init();
```

```
67 }
68
69 void
70 Router::wakeup()
71 {
72     DPRINTF(RubyNetwork, "Router %d woke up\n", m_id);
73
74     // changes by Soultana Ellinidou-SDNoC
75     //**************************************
76     // Update Time
77
78     schedule_wakeup(Cycles(tau));
79     std::vector<InputUnit *> input_tmp = get_inputUnit_ref();
80     std::vector<OutputUnit *> output_tmp = get_outputUnit_ref();
81
82     // // Technique init phase
83     // if (curTick() == tau)
84     // {
85     //     for (auto elem : input_tmp)
86     //         elem->set_state_rate_saved(0);
87     // }
88     // //
89
90     if (curTick() % tau == 0 && tau != 0) {
91         for (auto elem : input_tmp)
92         {
93             double dtau = (double) tau;
94             double old_state = elem->get_state_rate_saved();
95             double new_state = elem->get_state_rate() / dtau;
96             double update = (new_state + alpha * old_state) / (1.0 + alpha);
97             elem->set_state_rate_saved(update);
98             elem->set_state_rate(0.0);
99
100        }
101
102        for (auto elem : output_tmp)
103        {
104            // penality
105            elem->set_penality(0.0);
106        }
107    }
108    //**************************************
109
110    // check for incoming flits
111    for (int inport = 0; inport < m_input_unit.size(); inport++) {
112        m_input_unit[inport]->wakeup();
113    }
114
115    // check for incoming credits
116    // Note: the credit update is happening before SA
117    // buffer turnaround time =
118    //     credit traversal (1-cycle) + SA (1-cycle) + Link Traversal (1-cycle)
119    // if we want the credit update to take place after SA, this loop should
120    // be moved after the SA request
121    for (int outport = 0; outport < m_output_unit.size(); outport++) {
122        m_output_unit[outport]->wakeup();
123    }
124
125    // Switch Allocation
126    m_sw_alloc->wakeup();
127
128    // Switch Traversal
129    m_switch->wakeup();
130 }
131
132 void
133 Router::addInPort(PortDirection inport_dirn,
134                   NetworkLink *in_link, CreditLink *credit_link)
135 {
136     int port_num = m_input_unit.size();
137     InputUnit *input_unit = new InputUnit(port_num, inport_dirn, this);
138     input_unit->set_in_link(in_link);
139     input_unit->set_credit_link(credit_link);
140     in_link->setLinkConsumer(this);
141     credit_link->setSourceQueue(input_unit->getCreditQueue());
142
143     m_input_unit.push_back(input_unit);
144
145     m_routing_unit->addInDirection(inport_dirn, port_num);
146
```

```
147     // // changes by Soultana Ellinidou-SDNoC
148     get_net_ptr()->set_map_link_inport(in_link, input_unit);
149 }
150
151 void
152 Router::addOutPort(PortDirection outport_dirn,
153                    NetworkLink *out_link,
154                    const NetDest& routing_table_entry, int link_weight,
155                    CreditLink *credit_link)
156 {
157     int port_num = m_output_unit.size();
158     OutputUnit *output_unit = new OutputUnit(port_num, outport_dirn, this);
159
160     output_unit->set_out_link(out_link);
161     output_unit->set_credit_link(credit_link);
162     credit_link->setLinkConsumer(this);
163     out_link->setSourceQueue(output_unit->getOutQueue());
164
165     m_output_unit.push_back(output_unit);
166
167     m_routing_unit->addRoute(routing_table_entry);
168     m_routing_unit->addWeight(link_weight);
169     m_routing_unit->addOutDirection(outport_dirn, port_num);
170
171
172     // // changes by Soultana Ellinidou-SDNoC
173     output_unit->set_dst_router(map_link_node[out_link]);
174     set_map_direction_out(outport_dirn, output_unit);
175     set_map_dst_out(map_link_node[out_link], output_unit);
176
177     if (outport_dirn != "Local" && outport_dirn != "to_node_src" && outport_dirn
       != "to_sdn_src")
178         get_net_ptr()->set_matrix(get_id(), map_link_node[out_link], output_unit
      );
179 }
180
181 PortDirection
182 Router::getOutportDirection(int outport)
183 {
184     return m_output_unit[outport]->get_direction();
185 }
186
187 PortDirection
188 Router::getInportDirection(int inport)
189 {
190     return m_input_unit[inport]->get_direction();
191 }
192
193 int
194 Router::route_compute(RouteInfo route, int inport, PortDirection inport_dirn)
195 {
196     return m_routing_unit->outportCompute(route, inport, inport_dirn);
197 }
198
199 //*************************************
200 // // changes by Soultana Ellinidou-SDNoC
201 int
202 Router::route_back(RouteInfo route)
203 {
204     return m_routing_unit->outportBack(route);
205 }
206
207 int
208 Router::route_back_hack(RouteInfo route)
209 {
210     return m_routing_unit->outportBack_hack(route);
211 }
212
213 int
214 Router::route_controller(RouteInfo route)
215 {
216     return m_routing_unit->outportController(route);
217 }
218
219 void
220 Router::route_compute_SDN(RouteInfo route)
221 {
222     return m_routing_unit->SDN_outportCompute(route);
223 }
224
```

```
225 int
226 Router::route_compute_NoC(RouteInfo route)
227 {
228     return m_routing_unit->NoC_outportCompute(route);
229 }
230
231 //*************************************************
232
233 void
234 Router::grant_switch(int inport, flit *t_flit)
235 {
236     m_switch->update_sw_winner(inport, t_flit);
237 }
238
239 void
240 Router::schedule_wakeup(Cycles time)
241 {
242     // wake up after time cycles
243     scheduleEvent(time);
244 }
245
246 std::string
247 Router::getPortDirectionName(PortDirection direction)
248 {
249     // PortDirection is actually a string
250     // If not, then this function should add a switch
251     // statement to convert direction to a string
252     // that can be printed out
253     return direction;
254 }
255
256 void
257 Router::regStats()
258 {
259     BasicRouter::regStats();
260
261     m_buffer_reads
262         .name(name() + ".buffer_reads")
263         .flags(Stats::nozero)
264     ;
265
266     m_buffer_writes
267         .name(name() + ".buffer_writes")
268         .flags(Stats::nozero)
269     ;
270
271     m_crossbar_activity
272         .name(name() + ".crossbar_activity")
273         .flags(Stats::nozero)
274     ;
275
276     m_sw_input_arbiter_activity
277         .name(name() + ".sw_input_arbiter_activity")
278         .flags(Stats::nozero)
279     ;
280
281     m_sw_output_arbiter_activity
282         .name(name() + ".sw_output_arbiter_activity")
283         .flags(Stats::nozero)
284     ;
285 }
286
287 void
288 Router::collateStats()
289 {
290     for (int j = 0; j < m_virtual_networks; j++) {
291         for (int i = 0; i < m_input_unit.size(); i++) {
292             m_buffer_reads += m_input_unit[i]->get_buf_read_activity(j);
293             m_buffer_writes += m_input_unit[i]->get_buf_write_activity(j);
294         }
295     }
296
297     m_sw_input_arbiter_activity = m_sw_alloc->get_input_arbiter_activity();
298     m_sw_output_arbiter_activity = m_sw_alloc->get_output_arbiter_activity();
299     m_crossbar_activity = m_switch->get_crossbar_activity();
300 }
301
302 void
303 Router::resetStats()
304 {
```

```
305     for (int j = 0; j < m_virtual_networks; j++) {
306         for (int i = 0; i < m_input_unit.size(); i++) {
307             m_input_unit[i]->resetStats();
308         }
309     }
310
311     m_switch->resetStats();
312     m_sw_alloc->resetStats();
313 }
314
315 void
316 Router::printFaultVector(ostream& out)
317 {
318     int temperature_celcius = BASELINE_TEMPERATURE_CELCIUS;
319     int num_fault_types = m_network_ptr->fault_model->number_of_fault_types;
320     float fault_vector[num_fault_types];
321     get_fault_vector(temperature_celcius, fault_vector);
322     out << "Router-" << m_id << " fault vector: " << endl;
323     for (int fault_type_index = 0; fault_type_index < num_fault_types;
324          fault_type_index++) {
325         out << " - probability of (";
326         out <<
327         m_network_ptr->fault_model->fault_type_to_string(fault_type_index);
328         out << ") = ";
329         out << fault_vector[fault_type_index] << endl;
330     }
331 }
332
333 void
334 Router::printAggregateFaultProbability(std::ostream& out)
335 {
336     int temperature_celcius = BASELINE_TEMPERATURE_CELCIUS;
337     float aggregate_fault_prob;
338     get_aggregate_fault_probability(temperature_celcius,
339                                     &aggregate_fault_prob);
340     out << "Router-" << m_id << " fault probability: ";
341     out << aggregate_fault_prob << endl;
342 }
343
344 uint32_t
345 Router::functionalWrite(Packet *pkt)
346 {
347     uint32_t num_functional_writes = 0;
348     num_functional_writes += m_switch->functionalWrite(pkt);
349
350     for (uint32_t i = 0; i < m_input_unit.size(); i++) {
351         num_functional_writes += m_input_unit[i]->functionalWrite(pkt);
352     }
353
354     for (uint32_t i = 0; i < m_output_unit.size(); i++) {
355         num_functional_writes += m_output_unit[i]->functionalWrite(pkt);
356     }
357
358     return num_functional_writes;
359 }
360
361 Router *
362 GarnetRouterParams::create()
363 {
364     return new Router(this);
365 }
```

Listing A.2: Router.cc

```
1  #include "mem/ruby/network/garnet2.0/GarnetNetwork.hh"
2
3  #include <cassert>
4
5  #include "base/cast.hh"
6  #include "base/stl_helpers.hh"
7  #include "mem/ruby/common/NetDest.hh"
8  #include "mem/ruby/network/MessageBuffer.hh"
9  #include "mem/ruby/network/garnet2.0/CommonTypes.hh"
10 #include "mem/ruby/network/garnet2.0/CreditLink.hh"
11 #include "mem/ruby/network/garnet2.0/GarnetLink.hh"
12 #include "mem/ruby/network/garnet2.0/NetworkInterface.hh"
13 #include "mem/ruby/network/garnet2.0/NetworkLink.hh"
14 #include "mem/ruby/network/garnet2.0/Router.hh"
```

```
15 #include "mem/ruby/system/RubySystem.hh"
16
17 // changes by Soultana Ellinidou-SDNoC
18 #include "mem/ruby/network/garnet2.0/OutputUnit.hh"
19 #include "mem/ruby/network/garnet2.0/InputUnit.hh"
20 //
21
22 using namespace std;
23 using m5::stl_helpers::deletePointers;
24
25 /*
26  * GarnetNetwork sets up the routers and links and collects stats.
27  * Default parameters (GarnetNetwork.py) can be overwritten from command line
28  * (see configs/network/Network.py)
29  */
30
31 GarnetNetwork::GarnetNetwork(const Params *p)
32     : Network(p)
33 {
34     m_num_rows = p->num_rows;
35     m_ni_flit_size = p->ni_flit_size;
36     m_vcs_per_vnet = p->vcs_per_vnet;
37     m_buffers_per_data_vc = p->buffers_per_data_vc;
38     m_buffers_per_ctrl_vc = p->buffers_per_ctrl_vc;
39     m_routing_algorithm = p->routing_algorithm;
40
41     m_enable_fault_model = p->enable_fault_model;
42     if (m_enable_fault_model)
43         fault_model = p->fault_model;
44
45     m_vnet_type.resize(m_virtual_networks);
46
47     for (int i = 0 ; i < m_virtual_networks ; i++) {
48         if (m_vnet_type_names[i] == "response")
49             m_vnet_type[i] = DATA_VNET_; // carries data (and ctrl) packets
50         else
51             m_vnet_type[i] = CTRL_VNET_; // carries only ctrl packets
52     }
53
54     // record the routers
55     for (vector<BasicRouter*>::const_iterator i =  p->routers.begin();
56          i != p->routers.end(); ++i) {
57         Router* router = safe_cast<Router*>(*i);
58         m_routers.push_back(router);
59
60         // initialize the router's network pointers
61         router->init_net_ptr(this);
62     }
63
64     // record the network interfaces
65     for (vector<ClockedObject*>::const_iterator i = p->netifs.begin();
66          i != p->netifs.end(); ++i) {
67         NetworkInterface *ni = safe_cast<NetworkInterface *>(*i);
68         m_nis.push_back(ni);
69         ni->init_net_ptr(this);
70     }
71
72     //**************************************************
73     // // changes by Soultana Ellinidou-SDNoC
74
75     // record the SDN controller
76     Router* tmp = safe_cast<Router*>(p->sdnc[0]);
77     m_sdnc.push_back(tmp);
78     m_sdnc[0]->init_net_ptr(this);
79     std::cout << "GarnetNetwork.cc : Controller ID: " << m_sdnc[0]->get_id() <<
       std::endl;
80
81     // record NI controller
82     NetworkInterface *ni = safe_cast<NetworkInterface *>(p->NI_c[0]);
83     m_NIc.push_back(ni);
84     ni->init_net_ptr(this);
85     std::cout << "GarnetNetwork.cc : NI of controller set" << std::endl;
86
87
88     // init the matrix
89     init_matrix_pntr();
90     init_matrix_label();
91
92
93     //**************************************************
```

```
94 }
95
96 void
97 GarnetNetwork::init()
98 {
99     Network::init();
100
101     for (int i=0; i < m_nodes; i++) {
102         m_nis[i]->addNode(m_toNetQueues[i], m_fromNetQueues[i]);
103     }
104
105     // The topology pointer should have already been initialized in the
106     // parent network constructor
107     assert(m_topology_ptr != NULL);
108     m_topology_ptr->createLinks(this);
109
110     // Initialize topology specific parameters
111     if (getNumRows() > 0) {
112         // Only for Mesh topology
113         // m_num_rows and m_num_cols are only used for
114         // implementing XY or custom routing in RoutingUnit.cc
115         m_num_rows = getNumRows();
116         m_num_cols = m_routers.size() / m_num_rows;
117         assert(m_num_rows * m_num_cols == m_routers.size());
118     } else {
119         m_num_rows = -1;
120         m_num_cols = -1;
121     }
122
123     // FaultModel: declare each router to the fault model
124     if (isFaultModelEnabled()) {
125         for (vector<Router*>::const_iterator i= m_routers.begin();
126              i != m_routers.end(); ++i) {
127             Router* router = safe_cast<Router*>(*i);
128             int router_id M5_VAR_USED =
129                 fault_model->declare_router(router->get_num_inports(),
130                                             router->get_num_outports(),
131                                             router->get_vc_per_vnet(),
132                                             getBuffersPerDataVC(),
133                                             getBuffersPerCtrlVC());
134             assert(router_id == router->get_id());
135             router->printAggregateFaultProbability(cout);
136             router->printFaultVector(cout);
137         }
138
139         // SDN
140         Router* router = safe_cast<Router*>(m_sdnc[0]);
141         int router_id M5_VAR_USED = fault_model->declare_router(router->
    get_num_inports(),
142                                                                 router->
    get_num_outports(),
143                                                                 router->
    get_vc_per_vnet(),
144
    getBuffersPerDataVC(),
145
    getBuffersPerCtrlVC());
146         assert(router_id == router->get_id());
147         router->printAggregateFaultProbability(cout);
148         router->printFaultVector(cout);
149     }
150 }
151
152 GarnetNetwork::~GarnetNetwork()
153 {
154     deletePointers(m_routers);
155     deletePointers(m_nis);
156     deletePointers(m_networklinks);
157     deletePointers(m_creditlinks);
158 }
159
160 /*
161  * This function creates a link from the Network Interface (NI)
162  * into the Network.
163  * It creates a Network Link from the NI to a Router and a Credit Link from
164  * the Router to the NI
165 */
166
167 void
168 GarnetNetwork::makeExtInLink(NodeID src, SwitchID dest, BasicLink* link,
```

```
169                                 const NetDest& routing_table_entry)
170 {
171     assert(src < m_nodes);
172
173     GarnetExtLink* garnet_link = safe_cast<GarnetExtLink*>(link);
174
175     // GarnetExtLink is bi-directional
176     NetworkLink* net_link = garnet_link->m_network_links[LinkDirection_In];
177     net_link->setType(EXT_IN_);
178     CreditLink* credit_link = garnet_link->m_credit_links[LinkDirection_In];
179
180     m_networklinks.push_back(net_link);
181     m_creditlinks.push_back(credit_link);
182
183     PortDirection dst_inport_dirn = "Local";
184     m_routers[dest]->addInPort(dst_inport_dirn, net_link, credit_link);
185     m_nis[src]->addOutPort(net_link, credit_link, dest);
186 }
187
188 /*
189  * This function creates a link from the Network to a NI.
190  * It creates a Network Link from a Router to the NI and
191  * a Credit Link from NI to the Router
192 */
193
194 void
195 GarnetNetwork::makeExtOutLink(SwitchID src, NodeID dest, BasicLink* link,
196                               const NetDest& routing_table_entry)
197 {
198     assert(dest < m_nodes);
199     assert(src < m_routers.size());
200     assert(m_routers[src] != NULL);
201
202     GarnetExtLink* garnet_link = safe_cast<GarnetExtLink*>(link);
203
204     // GarnetExtLink is bi-directional
205     NetworkLink* net_link = garnet_link->m_network_links[LinkDirection_Out];
206     net_link->setType(EXT_OUT_);
207     CreditLink* credit_link = garnet_link->m_credit_links[LinkDirection_Out];
208
209     m_networklinks.push_back(net_link);
210     m_creditlinks.push_back(credit_link);
211
212     PortDirection src_outport_dirn = "Local";
213     m_routers[src]->addOutPort(src_outport_dirn, net_link,
214                                routing_table_entry,
215                                link->m_weight, credit_link);
216     m_nis[dest]->addInPort(net_link, credit_link);
217 }
218
219 /*
220  * This function creates an internal network link between two routers.
221  * It adds both the network link and an opposite credit link.
222 */
223
224 void
225 GarnetNetwork::makeInternalLink(SwitchID src, SwitchID dest, BasicLink* link,
226                                 const NetDest& routing_table_entry,
227                                 PortDirection src_outport_dirn,
228                                 PortDirection dst_inport_dirn)
229 {
230     GarnetIntLink* garnet_link = safe_cast<GarnetIntLink*>(link);
231
232     // GarnetIntLink is unidirectional
233     NetworkLink* net_link = garnet_link->m_network_link;
234     net_link->setType(INT_);
235     CreditLink* credit_link = garnet_link->m_credit_link;
236
237     m_networklinks.push_back(net_link);
238     m_creditlinks.push_back(credit_link);
239
240     /// changes by Soultana Ellinidou-SDNoC
241
242     //
243
244     if (src == getNumRouters())
245     {
246         m_sdnc[0]->set_map_link_node(net_link, m_routers[dest]->get_id());
247         m_routers[dest]->addInPort(dst_inport_dirn, net_link, credit_link);
248         m_sdnc[0]->addOutPort(src_outport_dirn, net_link,
```

```
249                                 routing_table_entry,
250                                 link->m_weight, credit_link);
251     }
252
253     else if (dest == getNumRouters())
254     {
255         m_routers[src]->set_map_link_node(net_link, m_sdnc[0]->get_id());
256         m_sdnc[0]->addInPort(dst_inport_dirn, net_link, credit_link);
257         m_routers[src]->addOutPort(src_outport_dirn, net_link,
258                                    routing_table_entry,
259                                    link->m_weight, credit_link);
260     }
261
262     else
263     {
264         m_routers[src]->set_map_link_node(net_link, m_routers[dest]->get_id());
265         m_routers[dest]->addInPort(dst_inport_dirn, net_link, credit_link);
266         m_routers[src]->addOutPort(src_outport_dirn, net_link,
267                                    routing_table_entry,
268                                    link->m_weight, credit_link);
269     }
270 }
271
272 // Total routers in the network
273 int
274 GarnetNetwork::getNumRouters()
275 {
276     return m_routers.size();
277 }
278
279 // Get ID of router connected to a NI.
280 int
281 GarnetNetwork::get_router_id(int ni)
282 {
283     return m_nis[ni]->get_router_id();
284 }
285
286 void
287 GarnetNetwork::regStats()
288 {
289     Network::regStats();
290
291     // Packets
292     m_packets_received
293         .init(m_virtual_networks)
294         .name(name() + ".packets_received")
295         .flags(Stats::pdf | Stats::total | Stats::nozero | Stats::oneline)
296         ;
297
298     m_packets_injected
299         .init(m_virtual_networks)
300         .name(name() + ".packets_injected")
301         .flags(Stats::pdf | Stats::total | Stats::nozero | Stats::oneline)
302         ;
303
304     m_packet_network_latency
305         .init(m_virtual_networks)
306         .name(name() + ".packet_network_latency")
307         .flags(Stats::oneline)
308         ;
309
310     m_packet_queueing_latency
311         .init(m_virtual_networks)
312         .name(name() + ".packet_queueing_latency")
313         .flags(Stats::oneline)
314         ;
315
316     for (int i = 0; i < m_virtual_networks; i++) {
317         m_packets_received.subname(i, csprintf("vnet-%i", i));
318         m_packets_injected.subname(i, csprintf("vnet-%i", i));
319         m_packet_network_latency.subname(i, csprintf("vnet-%i", i));
320         m_packet_queueing_latency.subname(i, csprintf("vnet-%i", i));
321     }
322
323     m_avg_packet_vnet_latency
324         .name(name() + ".average_packet_vnet_latency")
325         .flags(Stats::oneline);
326     m_avg_packet_vnet_latency =
327         m_packet_network_latency / m_packets_received;
328
```

```
329     m_avg_packet_vqueue_latency
330         .name(name() + ".average_packet_vqueue_latency")
331         .flags(Stats::oneline);
332     m_avg_packet_vqueue_latency =
333         m_packet_queueing_latency / m_packets_received;
334
335     m_avg_packet_network_latency
336         .name(name() + ".average_packet_network_latency");
337     m_avg_packet_network_latency =
338         sum(m_packet_network_latency) / sum(m_packets_received);
339
340     m_avg_packet_queueing_latency
341         .name(name() + ".average_packet_queueing_latency");
342     m_avg_packet_queueing_latency
343         = sum(m_packet_queueing_latency) / sum(m_packets_received);
344
345     m_avg_packet_latency
346         .name(name() + ".average_packet_latency");
347     m_avg_packet_latency
348         = m_avg_packet_network_latency + m_avg_packet_queueing_latency;
349
350     // Flits
351     m_flits_received
352         .init(m_virtual_networks)
353         .name(name() + ".flits_received")
354         .flags(Stats::pdf | Stats::total | Stats::nozero | Stats::oneline)
355         ;
356
357     m_flits_injected
358         .init(m_virtual_networks)
359         .name(name() + ".flits_injected")
360         .flags(Stats::pdf | Stats::total | Stats::nozero | Stats::oneline)
361         ;
362
363     m_flit_network_latency
364         .init(m_virtual_networks)
365         .name(name() + ".flit_network_latency")
366         .flags(Stats::oneline)
367         ;
368
369     m_flit_queueing_latency
370         .init(m_virtual_networks)
371         .name(name() + ".flit_queueing_latency")
372         .flags(Stats::oneline)
373         ;
374
375     for (int i = 0; i < m_virtual_networks; i++) {
376         m_flits_received.subname(i, csprintf("vnet-%i", i));
377         m_flits_injected.subname(i, csprintf("vnet-%i", i));
378         m_flit_network_latency.subname(i, csprintf("vnet-%i", i));
379         m_flit_queueing_latency.subname(i, csprintf("vnet-%i", i));
380     }
381
382     m_avg_flit_vnet_latency
383         .name(name() + ".average_flit_vnet_latency")
384         .flags(Stats::oneline);
385     m_avg_flit_vnet_latency = m_flit_network_latency / m_flits_received;
386
387     m_avg_flit_vqueue_latency
388         .name(name() + ".average_flit_vqueue_latency")
389         .flags(Stats::oneline);
390     m_avg_flit_vqueue_latency =
391         m_flit_queueing_latency / m_flits_received;
392
393     m_avg_flit_network_latency
394         .name(name() + ".average_flit_network_latency");
395     m_avg_flit_network_latency =
396         sum(m_flit_network_latency) / sum(m_flits_received);
397
398     m_avg_flit_queueing_latency
399         .name(name() + ".average_flit_queueing_latency");
400     m_avg_flit_queueing_latency =
401         sum(m_flit_queueing_latency) / sum(m_flits_received);
402
403     m_avg_flit_latency
404         .name(name() + ".average_flit_latency");
405     m_avg_flit_latency =
406         m_avg_flit_network_latency + m_avg_flit_queueing_latency;
407
408
```

```
409     // Hops
410     m_avg_hops.name(name() + ".average_hops");
411     m_avg_hops = m_total_hops / sum(m_flits_received);
412
413     // Links
414     m_total_ext_in_link_utilization
415         .name(name() + ".ext_in_link_utilization");
416     m_total_ext_out_link_utilization
417         .name(name() + ".ext_out_link_utilization");
418     m_total_int_link_utilization
419         .name(name() + ".int_link_utilization");
420     m_average_link_utilization
421         .name(name() + ".avg_link_utilization");
422
423     m_average_vc_load
424         .init(m_virtual_networks * m_vcs_per_vnet)
425         .name(name() + ".avg_vc_load")
426         .flags(Stats::pdf | Stats::total | Stats::nozero | Stats::oneline)
427         ;
428 }
429
430 void
431 GarnetNetwork::collateStats()
432 {
433     RubySystem *rs = params()->ruby_system;
434     double time_delta = double(curCycle() - rs->getStartCycle());
435
436     for (int i = 0; i < m_networklinks.size(); i++) {
437         link_type type = m_networklinks[i]->getType();
438         int activity = m_networklinks[i]->getLinkUtilization();
439
440         if (type == EXT_IN_)
441             m_total_ext_in_link_utilization += activity;
442         else if (type == EXT_OUT_)
443             m_total_ext_out_link_utilization += activity;
444         else if (type == INT_)
445             m_total_int_link_utilization += activity;
446
447         m_average_link_utilization +=
448             (double(activity) / time_delta);
449
450         vector<unsigned int> vc_load = m_networklinks[i]->getVcLoad();
451         for (int j = 0; j < vc_load.size(); j++) {
452             m_average_vc_load[j] += ((double)vc_load[j] / time_delta);
453         }
454     }
455
456     // Ask the routers to collate their statistics
457     for (int i = 0; i < m_routers.size(); i++) {
458         m_routers[i]->collateStats();
459     }
460 }
461
462 void
463 GarnetNetwork::print(ostream& out) const
464 {
465     out << "[GarnetNetwork]";
466 }
467
468 GarnetNetwork *
469 GarnetNetworkParams::create()
470 {
471     return new GarnetNetwork(this);
472 }
473
474 uint32_t
475 GarnetNetwork::functionalWrite(Packet *pkt)
476 {
477     uint32_t num_functional_writes = 0;
478
479     for (unsigned int i = 0; i < m_routers.size(); i++) {
480         num_functional_writes += m_routers[i]->functionalWrite(pkt);
481     }
482
483     for (unsigned int i = 0; i < m_nis.size(); ++i) {
484         num_functional_writes += m_nis[i]->functionalWrite(pkt);
485     }
486
487     for (unsigned int i = 0; i < m_networklinks.size(); ++i) {
488         num_functional_writes += m_networklinks[i]->functionalWrite(pkt);
```

```
489     }
490
491     return num_functional_writes;
492 }
493
494 // changes by Soultana Ellinidou-SDNoC
495 void
496 GarnetNetwork::init_matrix_pntr()
497 {
498     int N = getNumRouters();
499     for (int i = 0; i < N; i++)
500     {
501         for (int j = 0; j < N; j++)
502         {
503             if (i == j)
504                 //m_matrix.push_back(std::make_pair("C", nullptr));
505                 m_matrix_pntr.push_back(nullptr);
506             else
507                 //m_matrix.push_back(std::make_pair("0", nullptr));
508                 m_matrix_pntr.push_back(nullptr);
509         }
510     }
511 }
512
513 void
514 GarnetNetwork::init_matrix_label()
515 {
516     int N = getNumRouters();
517     for (int i = 0; i < N; i++)
518     {
519         for (int j = 0; j < N; j++)
520         {
521             if (i == j)
522                 m_matrix_label.push_back(CURRENT);
523             else
524                 m_matrix_label.push_back(ZERO);
525         }
526     }
527 }
528
529 void
530 GarnetNetwork::set_matrix(int router_id, int router_dst, OutputUnit* out)
531 {
532     int N = getNumRouters();
533     m_matrix_pntr[router_id * N + router_dst] = out;
534
535     std::string portdir = (std::string) out->get_direction();
536     int label;
537     if (portdir == "U")
538         label = UP;
539     else if (portdir == "D")
540         label = DOWN;
541     else if (portdir == "North")
542         label = NORTH;
543     else if (portdir == "South")
544         label = SOUTH;
545     else if (portdir == "East")
546         label = EAST;
547     else
548         label = WEST;
549     m_matrix_label[router_id * N + router_dst] = label;
550 }
```

Listing A.3: GarnetNetwork.cc

```
1  #include "mem/ruby/network/garnet2.0/InputUnit.hh"
2
3  #include "base/stl_helpers.hh"
4  #include "debug/RubyNetwork.hh"
5  #include "mem/ruby/network/garnet2.0/Credit.hh"
6  #include "mem/ruby/network/garnet2.0/Router.hh"
7
8  using namespace std;
9  using m5::stl_helpers::deletePointers;
10
11 InputUnit::InputUnit(int id, PortDirection direction, Router *router)
12             : Consumer(router)
13 {
```

```
14      m_id = id;
15      m_direction = direction;
16      m_router = router;
17      m_num_vcs = m_router->get_num_vcs();
18      m_vc_per_vnet = m_router->get_vc_per_vnet();
19
20
21      m_num_buffer_reads.resize(m_num_vcs/m_vc_per_vnet);
22      m_num_buffer_writes.resize(m_num_vcs/m_vc_per_vnet);
23
24      for (int i = 0; i < m_num_buffer_reads.size(); i++) {
25          m_num_buffer_reads[i] = 0;
26          m_num_buffer_writes[i] = 0;
27      }
28
29      creditQueue = new flitBuffer();
30      // Instantiating the virtual channels
31      m_vcs.resize(m_num_vcs);
32      for (int i=0; i < m_num_vcs; i++) {
33          m_vcs[i] = new VirtualChannel(i);
34      }
35
36      //changes by Soultana Ellinidou-SDNoC
37      state_rate = 0.0;
38      state_rate_saved = 0.0;
39
40      std::ifstream reader;
41      reader.open("/home/gaurav/gem5/var/alpha.txt");
42      if (!reader)
43          assert(0);
44      reader >> alpha;
45      //std::cout << "alpha: " <<  alpha << std::endl;
46      reader.close();
47
48      std::ifstream reader2;
49      reader2.open("/home/gaurav/gem5/var/tau.txt");
50      if (!reader2)
51          assert(0);
52      reader2 >> tau;
53      reader2.close();
54
55      std::ifstream reader3;
56      reader3.open("/home/gaurav/gem5/var/timeout.txt");
57      if (!reader3)
58          assert(0);
59      reader3 >> timeout;
60      reader3.close();
61
62  }
63
64  InputUnit::~InputUnit()
65  {
66      delete creditQueue;
67      deletePointers(m_vcs);
68  }
69
70  /*
71   * The InputUnit wakeup function reads the input flit from its input link.
72   * Each flit arrives with an input VC.
73   * For HEAD/HEAD_TAIL flits, performs route computation,
74   * and updates route in the input VC.
75   * The flit is buffered for (m_latency - 1) cycles in the input VC
76   * and marked as valid for SwitchAllocation starting that cycle.
77   *
78   */
79
80  void
81  InputUnit::wakeup()
82  {
83      // bool updateTime = false;
84      flit *t_flit;
85      if (m_in_link->isReady(m_router->curCycle())) {
86
87          t_flit = m_in_link->consumeLink();
88          int vc = t_flit->get_vc();
89          t_flit->increment_hops(); // for stats
90
91          if ((t_flit->get_type() == HEAD_) ||
92              (t_flit->get_type() == HEAD_TAIL_)) {
93
```

```
94              assert(m_vcs[vc]->get_state() == IDLE_);
95              set_vc_active(vc, m_router->curCycle());
96
97              // Route computation for this vc
98
99              //*************************************************
100             // changes by Soultana Ellinidou-SDNoC
101
102             // NoC router
103             if (m_router->get_id() < m_router->get_net_ptr()->getNumRouters())
104             {
105                 std::pair<int, int> pair = std::make_pair(t_flit->get_route().
    src_router, t_flit->get_route().dest_router);
106                 std::map<std::pair<int, int>, int> flow = m_router->
    get_flow_table();
107                 bool tmp = !(flow.count(pair) > 0);
108                 bool tmp2 = m_router->get_flow_table().size() == 0;
109
110                 // Send packets to local node
111                 if (m_router->get_id() == t_flit->get_route().dest_router)
112                 {
113                     int outport = m_router->route_compute_NoC(t_flit->get_route
    ());
114                     grant_outport(vc, outport);
115                 }
116
117                 // Send the packet to the controller if no entry
118                 else if (tmp == true || tmp2 == true)
119                 {
120                     int outport = m_router->route_controller(t_flit->get_route()
    );
121                     grant_outport(vc, outport);
122                 }
123
124                 // Use flow table
125                 else
126                 {
127                     if (curTick() >= m_router->get_entry_timeout(pair))
128                     {
129                         int outport = m_router->route_controller(t_flit->
    get_route());
130                         grant_outport(vc, outport);
131                     }
132
133                     else
134                     {
135                         int outport = m_router->get_entry(pair);
136                         m_router->set_flow_timeout(pair, curTick() + timeout);
137                         grant_outport(vc, outport);
138                     }
139                 }
140             }
141
142             // SDN controller
143             else
144             {
145                 // Route computation
146                 m_router->route_compute_SDN(t_flit->get_route());
147
148                 // Send back to the node
149                 int back_to_node = m_router->route_back(t_flit->get_route());
150                 grant_outport(vc, back_to_node);
151             }
152             //*************************************************
153
154         } else {
155             assert(m_vcs[vc]->get_state() == ACTIVE_);
156         }
157
158         //*************************************************
159
160         // Buffer the flit
161         m_vcs[vc]->insertFlit(t_flit);
162
163         int vnet = vc/m_vc_per_vnet;
164         m_num_buffer_writes[vnet]++;
165         m_num_buffer_reads[vnet]++;
166
167         //*********************************
168         //changes by Soultana Ellinidou-SDNoC
```

```
169
170         if (vnet == 2)
171             state_rate++;
172
173         //********************************
174
175         Cycles pipe_stages = m_router->get_pipe_stages();
176         if (pipe_stages == 1) {
177             // 1-cycle router
178             // Flit goes for SA directly
179             t_flit->advance_stage(SA_, m_router->curCycle());
180         } else {
181             assert(pipe_stages > 1);
182             // Router delay is modeled by making flit wait in buffer for
183             // (pipe_stages cycles - 1) cycles before going for SA
184
185             Cycles wait_time = pipe_stages - Cycles(1);
186             t_flit->advance_stage(SA_, m_router->curCycle() + wait_time);
187
188             // Wakeup the router in that cycle to perform SA
189             m_router->schedule_wakeup(Cycles(wait_time));
190         }
191     }
192 }
193
194 // Send a credit back to upstream router for this VC.
195 // Called by SwitchAllocator when the flit in this VC wins the Switch.
196 void
197 InputUnit::increment_credit(int in_vc, bool free_signal, Cycles curTime)
198 {
199     Credit *t_credit = new Credit(in_vc, free_signal, curTime);
200     creditQueue->insert(t_credit);
201     m_credit_link->scheduleEventAbsolute(m_router->clockEdge(Cycles(1)));
202 }
203
204
205 uint32_t
206 InputUnit::functionalWrite(Packet *pkt)
207 {
208     uint32_t num_functional_writes = 0;
209     for (int i=0; i < m_num_vcs; i++) {
210         num_functional_writes += m_vcs[i]->functionalWrite(pkt);
211     }
212
213     return num_functional_writes;
214 }
215
216 void
217 InputUnit::resetStats()
218 {
219     for (int j = 0; j < m_num_buffer_reads.size(); j++) {
220         m_num_buffer_reads[j] = 0;
221         m_num_buffer_writes[j] = 0;
222     }
223     state_rate = 0;
224 }
```

Listing A.4: InputUnit.cc

```
1  #include "mem/ruby/network/garnet2.0/OutputUnit.hh"
2
3  #include "base/stl_helpers.hh"
4  #include "debug/RubyNetwork.hh"
5  #include "mem/ruby/network/garnet2.0/Credit.hh"
6  #include "mem/ruby/network/garnet2.0/Router.hh"
7
8  using namespace std;
9  using m5::stl_helpers::deletePointers;
10
11 OutputUnit::OutputUnit(int id, PortDirection direction, Router *router)
12     : Consumer(router)
13 {
14     m_id = id;
15     m_direction = direction;
16     m_router = router;
17     m_num_vcs = m_router->get_num_vcs();
18     m_vc_per_vnet = m_router->get_vc_per_vnet();
19     m_out_buffer = new flitBuffer();
```

```
20
21      for (int i = 0; i < m_num_vcs; i++) {
22          m_outvc_state.push_back(new OutVcState(i, m_router->get_net_ptr()));
23      }
24  }
25
26  OutputUnit::~OutputUnit()
27  {
28      delete m_out_buffer;
29      deletePointers(m_outvc_state);
30  }
31
32  void
33  OutputUnit::decrement_credit(int out_vc)
34  {
35      DPRINTF(RubyNetwork, "Router %d OutputUnit %d decrementing credit for "
36              "outvc %d at time: %lld\n",
37              m_router->get_id(), m_id, out_vc, m_router->curCycle());
38
39      m_outvc_state[out_vc]->decrement_credit();
40  }
41
42  void
43  OutputUnit::increment_credit(int out_vc)
44  {
45      DPRINTF(RubyNetwork, "Router %d OutputUnit %d incrementing credit for "
46              "outvc %d at time: %lld\n",
47              m_router->get_id(), m_id, out_vc, m_router->curCycle());
48
49      m_outvc_state[out_vc]->increment_credit();
50  }
51
52  // Check if the output VC (i.e., input VC at next router)
53  // has free credits (i..e, buffer slots).
54  // This is tracked by OutVcState
55  bool
56  OutputUnit::has_credit(int out_vc)
57  {
58      assert(m_outvc_state[out_vc]->isInState(ACTIVE_, m_router->curCycle()));
59      return m_outvc_state[out_vc]->has_credit();
60  }
61
62
63  // Check if the output port (i.e., input port at next router) has free VCs.
64  bool
65  OutputUnit::has_free_vc(int vnet)
66  {
67      int vc_base = vnet*m_vc_per_vnet;
68      for (int vc = vc_base; vc < vc_base + m_vc_per_vnet; vc++) {
69          if (is_vc_idle(vc, m_router->curCycle()))
70              return true;
71      }
72
73      return false;
74  }
75
76  // Assign a free output VC to the winner of Switch Allocation
77  int
78  OutputUnit::select_free_vc(int vnet)
79  {
80      int vc_base = vnet*m_vc_per_vnet;
81      for (int vc = vc_base; vc < vc_base + m_vc_per_vnet; vc++) {
82          if (is_vc_idle(vc, m_router->curCycle())) {
83              m_outvc_state[vc]->setState(ACTIVE_, m_router->curCycle());
84              return vc;
85          }
86      }
87
88      return -1;
89  }
90
91
92  // changes by Soultana Ellinidou-SDNoC
93  int
94  OutputUnit::get_free_slots_in(int vnet)
95  {
96      int free_slots_in = 0;
97      int vc_base = vnet * m_vc_per_vnet;
98      for (int vc = vc_base; vc < vc_base + m_vc_per_vnet; vc++) {
99          if (is_vc_idle(vc, m_router->curCycle())) {
```

```
100                 free_slots_in ++;
101             }
102         }
103         return free_slots_in;
104 }
105 //
106
107 /*
108  * The wakeup function of the OutputUnit reads the credit signal from the
109  * downstream router for the output VC (i.e., input VC at downstream router).
110  * It increments the credit count in the appropriate output VC state.
111  * If the credit carries is_free_signal as true,
112  * the output VC is marked IDLE.
113  */
114
115 void
116 OutputUnit::wakeup()
117 {
118     if (m_credit_link->isReady(m_router->curCycle())) {
119         Credit *t_credit = (Credit*) m_credit_link->consumeLink();
120         increment_credit(t_credit->get_vc());
121
122         if (t_credit->is_free_signal())
123         {
124             set_vc_state(IDLE_, t_credit->get_vc(), m_router->curCycle());
125         }
126
127         delete t_credit;
128     }
129 }
130
131 flitBuffer*
132 OutputUnit::getOutQueue()
133 {
134     return m_out_buffer;
135 }
136
137 void
138 OutputUnit::set_out_link(NetworkLink *link)
139 {
140     m_out_link = link;
141 }
142
143 void
144 OutputUnit::set_credit_link(CreditLink *credit_link)
145 {
146     m_credit_link = credit_link;
147 }
148
149 uint32_t
150 OutputUnit::functionalWrite(Packet *pkt)
151 {
152     return m_out_buffer->functionalWrite(pkt);
153 }
```

Listing A.5: OutputUnit.cc

```
1  import math
2  import m5
3  from m5.objects import *
4  from m5.defines import buildEnv
5  from m5.util import addToPath, fatal
6
7  def define_options(parser):
8      # By default, ruby uses the simple timing cpu
9      parser.set_defaults(cpu_type="TimingSimpleCPU")
10
11     parser.add_option("--topology", type="string", default="Crossbar",
12                       help="check configs/topologies for complete set")
13     parser.add_option("--mesh-rows", type="int", default=0,
14                       help="the number of rows in the mesh topology")
15     parser.add_option("--network", type="choice", default="simple",
16                       choices=['simple', 'garnet2.0'],
17                       help="'simple'|'garnet2.0'")
18     parser.add_option("--router-latency", action="store", type="int",
19                       default=1,
20                       help="""number of pipeline stages in the garnet router.
21                             Has to be >= 1.
```

```
22                              Can be over-ridden on a per router basis
23                              in the topology file.""")
24      parser.add_option("--link-latency", action="store", type="int", default=1,
25                        help="""latency of each link the simple/garnet networks.
26                              Has to be >= 1.
27                              Can be over-ridden on a per link basis
28                              in the topology file.""")
29      parser.add_option("--link-width-bits", action="store", type="int",
30                        default=128,
31                        help="width in bits for all links inside garnet.")
32      parser.add_option("--vcs-per-vnet", action="store", type="int", default=4,
33                        help="""number of virtual channels per virtual network
34                              inside garnet network.""")
35      parser.add_option("--routing-algorithm", action="store", type="int",
36                        default=0,
37                        help="""routing algorithm in network.
38                              0: weight-based table
39                              1: XY (for Mesh. see garnet2.0/RoutingUnit.cc)
40                              2: Custom (see garnet2.0/RoutingUnit.cc)""")
41      parser.add_option("--network-fault-model", action="store_true",
42                        default=False,
43                        help="""enable network fault model:
44                              see src/mem/ruby/network/fault_model/""")
45      parser.add_option("--garnet-deadlock-threshold", action="store",
46                        type="int", default=50000,
47                        help="network-level deadlock threshold.")
48
49
50  def create_network(options, ruby):
51
52      # Set the network classes based on the command line options
53      if options.network == "garnet2.0":
54          NetworkClass = GarnetNetwork
55          IntLinkClass = GarnetIntLink
56          ExtLinkClass = GarnetExtLink
57          RouterClass = GarnetRouter
58          InterfaceClass = GarnetNetworkInterface
59
60      else:
61          NetworkClass = SimpleNetwork
62          IntLinkClass = SimpleIntLink
63          ExtLinkClass = SimpleExtLink
64          RouterClass = Switch
65          InterfaceClass = None
66
67      # Instantiate the network object
68      # so that the controllers can connect to it.
69      # Adil Layach - sdnc & netifs_c
70      network = NetworkClass(ruby_system = ruby, topology = options.topology,
71              routers = [], ext_links = [], int_links = [], netifs = [], sdnc =
         [], NI_c = [])#, int_links_sdn = [])
72      return (network, IntLinkClass, ExtLinkClass, RouterClass, InterfaceClass)
73
74  def init_network(options, network, InterfaceClass):
75
76      if options.network == "garnet2.0":
77          network.num_rows = options.mesh_rows
78          network.vcs_per_vnet = options.vcs_per_vnet
79          network.ni_flit_size = options.link_width_bits / 8
80          network.routing_algorithm = options.routing_algorithm
81          network.garnet_deadlock_threshold = options.garnet_deadlock_threshold
82
83      if options.network == "simple":
84          network.setup_buffers()
85
86      if InterfaceClass != None:
87          netifs = [InterfaceClass(id=i) \
88                    for (i,n) in enumerate(network.ext_links)]
89          network.netifs = netifs
90
91          # changes by Soultana Ellinidou-SDNoC
92          NIc = [InterfaceClass(id=69)]
93          network.NI_c = NIc
94          #
95
96      if options.network_fault_model:
97          assert(options.network == "garnet2.0")
98          network.enable_fault_model = True
99          network.fault_model = FaultModel()
```

Listing A.6: Network.py

```
from m5.params import *
from m5.objects import *

from BaseTopology import SimpleTopology

# Creates a generic Mesh assuming an equal number of cache
# and directory controllers.
# XY routing is enforced (using link weights)
# to guarantee deadlock freedom.

class Mesh_XY(SimpleTopology):
    description='Mesh_XY'

    def __init__(self, controllers):
        self.nodes = controllers

    # Makes a generic mesh
    # assuming an equal number of cache and directory cntrls

    def makeTopology(self, options, network, IntLink, ExtLink, Router):
        nodes = self.nodes

        num_routers = options.num_cpus
        num_rows = options.mesh_rows

        # default values for link latency and router latency.
        # Can be over-ridden on a per link/router basis
        link_latency = options.link_latency # used by simple and garnet
        router_latency = options.router_latency # only used by garnet

        # There must be an evenly divisible number of cntrls to routers
        # Also, obviously the number or rows must be <= the number of routers
        cntrls_per_router, remainder = divmod(len(nodes), num_routers)
        assert(num_rows > 0 and num_rows <= num_routers)
        num_columns = int(num_routers / num_rows)
        assert(num_columns * num_rows == num_routers)

        # Create the routers in the mesh
        routers = [Router(router_id=i, latency = router_latency) \
            for i in range(num_routers)]
        network.routers = routers

        # link counter to set unique link ids
        link_count = 0

        # Add all but the remainder nodes to the list of nodes to be uniformly
        # distributed across the network.
        network_nodes = []
        remainder_nodes = []
        for node_index in xrange(len(nodes)):
            if node_index < (len(nodes) - remainder):
                network_nodes.append(nodes[node_index])
            else:
                remainder_nodes.append(nodes[node_index])

        # Connect each node to the appropriate router
        ext_links = []
        for (i, n) in enumerate(network_nodes):
            cntrl_level, router_id = divmod(i, num_routers)
            assert(cntrl_level < cntrls_per_router)
            ext_links.append(ExtLink(link_id=link_count, ext_node=n,
                                     int_node=routers[router_id],
                                     latency = link_latency))
            link_count += 1

        # Connect the remainding nodes to router 0.  These should only be
        # DMA nodes.
        for (i, node) in enumerate(remainder_nodes):
            assert(node.type == 'DMA_Controller')
            assert(i < remainder)
            ext_links.append(ExtLink(link_id=link_count, ext_node=node,
                                     int_node=routers[0],
                                     latency = link_latency))
            link_count += 1

```

```
76          network.ext_links = ext_links
77
78          # Create the mesh links.
79          int_links = []
80
81          # East output to West input links (weight = 1)
82          for row in xrange(num_rows):
83              for col in xrange(num_columns):
84                  if (col + 1 < num_columns):
85                      east_out = col + (row * num_columns)
86                      west_in = (col + 1) + (row * num_columns)
87                      int_links.append(IntLink(link_id=link_count,
88                                               src_node=routers[east_out],
89                                               dst_node=routers[west_in],
90                                               src_outport="East",
91                                               dst_inport="West",
92                                               latency = link_latency,
93                                               weight=1))
94                      link_count += 1
95
96          # West output to East input links (weight = 1)
97          for row in xrange(num_rows):
98              for col in xrange(num_columns):
99                  if (col + 1 < num_columns):
100                     east_in = col + (row * num_columns)
101                     west_out = (col + 1) + (row * num_columns)
102                     int_links.append(IntLink(link_id=link_count,
103                                              src_node=routers[west_out],
104                                              dst_node=routers[east_in],
105                                              src_outport="West",
106                                              dst_inport="East",
107                                              latency = link_latency,
108                                              weight=1))
109                     link_count += 1
110
111         # North output to South input links (weight = 2)
112         for col in xrange(num_columns):
113             for row in xrange(num_rows):
114                 if (row + 1 < num_rows):
115                     north_out = col + (row * num_columns)
116                     south_in = col + ((row + 1) * num_columns)
117                     int_links.append(IntLink(link_id=link_count,
118                                              src_node=routers[north_out],
119                                              dst_node=routers[south_in],
120                                              src_outport="North",
121                                              dst_inport="South",
122                                              latency = link_latency,
123                                              weight=2))
124                     link_count += 1
125
126         # South output to North input links (weight = 2)
127         for col in xrange(num_columns):
128             for row in xrange(num_rows):
129                 if (row + 1 < num_rows):
130                     north_in = col + (row * num_columns)
131                     south_out = col + ((row + 1) * num_columns)
132                     int_links.append(IntLink(link_id=link_count,
133                                              src_node=routers[south_out],
134                                              dst_node=routers[north_in],
135                                              src_outport="South",
136                                              dst_inport="North",
137                                              latency = link_latency,
138                                              weight=2))
139                     link_count += 1
140
141         #network.int_links = int_links
142
143         ########################################"
144         #changes by Soultana Ellinidou-SDNoC
145
146         # Create the controller
147         sdnc = [Router(router_id=num_routers, latency = router_latency)]
148         network.sdnc = sdnc
149
150         # Create the links to the controller
151         # int_links_sdn = []
152         for node in xrange(num_routers):
153             int_links.append(IntLink(link_id=link_count,
154                                      src_node=sdnc[0],
155                                      dst_node=routers[node],
```

```
156                                         src_outport="to_node_src",
157                                         dst_inport="to_node_dst",
158                                         latency = link_latency,
159                                         weight=1))
160             link_count += 1
161
162         for node in xrange(num_routers):
163             int_links.append(IntLink(link_id=link_count,
164                                         src_node=routers[node],
165                                         dst_node=sdnc[0],
166                                         src_outport="to_sdn_src",
167                                         dst_inport="to_sdn_dst",
168                                         latency = link_latency,
169                                         weight=1))
170             link_count += 1
171
172         network.int_links = int_links
173         ##################################################
174
175         print("number of int links: ", len(int_links))
176         print("number of ext links: ", len(ext_links))
```

Listing A.7: Mesh_XY.py

Bibliography

[Abd-El-Malek et al., 2005] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. (2005). Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74. ACM New York, NY, USA.

[Agarwal et al., 2009] Agarwal, N., Krishna, T., Peh, L.-S., and Jha, N. K. (2009). Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 33–42. IEEE.

[Al-Badi et al., 2009] Al-Badi, R., Al-Riyami, M., and Alzeidi, N. (2009). A parameterized noc simulator using omnet++. In *2009 International Conference on Ultra Modern Telecommunications & Workshops*, pages 1–7. IEEE.

[Ali et al., 2005] Ali, M., Welzl, M., and Hellebrand, S. (2005). A dynamic routing mechanism for network on chip. In *2005 NORCHIP*, pages 70–73. IEEE.

[Ancajas et al., 2014] Ancajas, D. M., Chakraborty, K., and Roy, S. (2014). Fort-nocs: Mitigating the threat of a compromised noc. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM.

[Arjomand and Sarbazi-Azad, 2008] Arjomand, M. and Sarbazi-Azad, H. (2008). Performance evaluation of butterfly on-chip network for mpsocs. In *2008 International SoC Design Conference*, volume 1, pages I–296. IEEE.

[Arunkumar et al., 2017] Arunkumar, A., Bolotin, E., Cho, B., Milic, U., Ebrahimi, E., Villa, O., Jaleel, A., Wu, C.-J., and Nellans, D. (2017). Mcm-gpu: Multi-chip-module gpus for continued performance scalability. *ACM SIGARCH Computer Architecture News*, 45(2):320–332. ACM.

[Atzori et al., 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer networks*, 54(15):2787–2805. Elsevier.

[Bahn and Bagherzadeh, 2008] Bahn, J. H. and Bagherzadeh, N. (2008). A generic traffic model for on-chip interconnection networks. *Network on Chip Architectures*, page 22.

[Benini and De Micheli, 2002] Benini, L. and De Micheli, G. (2002). Networks on chips: A new soc paradigm. *computer*, 35(1):70–78. IEEE.

[Benton et al., 2013] Benton, K., Camp, L. J., and Small, C. (2013). Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 151–152.

[Berestizshevsky et al., 2017] Berestizshevsky, K., Even, G., Fais, Y., and Ostrometzky, J. (2017). SDNoC: Software defined network on a chip. *Microprocessors and Microsystems*, 50:138–153. Elsevier.

[Bertozzi and Benini, 2004] Bertozzi, D. and Benini, L. (2004). Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems magazine*, 4(2):18–31. IEEE.

[Beyne and Manna, 2013] Beyne, E. and Manna, A. (2013). High-bandwidth chip-to-chip interfaces: 3d stacking, interposers and optical i/o. In *IMEC technology forum, Taiwan.*

[Bhunia and Tehranipoor, 2018] Bhunia, S. and Tehranipoor, M. (2018). *The Hardware Trojan War.* Springer.

[Binkert, 2020] Binkert, N. (2013 (accessed July 5, 2020)). *gem5 Simulator: Ruby Network Test.* http://www.m5sim.org/Ruby_Network_Test.

[Binkert et al., 2011] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7. ACM.

[Biswas et al., 2015] Biswas, A. K., Nandy, S., and Narayan, R. (2015). Router attack toward noc-enabled mpsoc and monitoring countermeasures against such threat. *Circuits, Systems, and Signal Processing*, 34(10):3241–3290. Springer.

[Bjerregaard and Mahadevan, 2006] Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1–es. ACM New York, NY, USA.

[Bjerregaard and Sparso, 2005a] Bjerregaard, T. and Sparso, J. (2005a). A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Design, Automation and Test in Europe*, pages 1226–1231. IEEE.

[Bjerregaard and Sparso, 2005b] Bjerregaard, T. and Sparso, J. (2005b). Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34–43. IEEE.

[Bjerregaard and Sparsø, 2006] Bjerregaard, T. and Sparsø, J. (2006). Implementation of guaranteed services in the mango clockless network-on-chip. *IEEE Proceedings-Computers and Digital Techniques*, 153(4):217–229. IET.

[Bland and Altman, 1996] Bland, J. M. and Altman, D. G. (1996). Statistics notes: measurement error. *Bmj*, 312(7047):1654. British Medical Journal Publishing Group.

[Blitzer et al., 2007] Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proceedings of the 45th annual meeting of the association of computational linguistics*, pages 440–447.

[Bolotin et al., 2004] Bolotin, E., Cidon, I., Ginosar, R., and Kolodny, A. (2004). Qnoc: Qos architecture and design process for network on chip. *Journal of systems architecture*, 50(2-3):105–128. Elsevier.

[Boneh and Franklin, 2001] Boneh, D. and Franklin, M. (2001). Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer.

[Boraten and Kodi, 2016] Boraten, T. and Kodi, A. K. (2016). Mitigation of denial of service attack with hardware trojans in noc architectures. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 1091–1100. IEEE.

[Bousdras et al., 2018] Bousdras, G., Quitin, F., and Milojevic, D. (2018). Template architectures for highly scalable, many-core heterogeneous soc: Could-of-chips. In *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE.

[Brooks and Yang, 2015] Brooks, M. and Yang, B. (2015). A man-in-the-middle attack against opendaylight sdn controller. In *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, pages 45–49. ACM.

[Cadence, 2018] Cadence (2018). Imec and Cadence Tape Out Industry's First 3nm Test Chip, Press Release. https://www.cadence.com/content/cadence-www/global/en_US/home/company/newsroom/press-releases/pr/2018/imec-and-cadence-tape-out-industry-s-first-3nm-test-chip.html. Accessed: April 2019.

[Castro et al., 1999] Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186.

[Catania et al., 2015] Catania, V., Mineo, A., Monteleone, S., Palesi, M., and Patti, D. (2015). Noxim: An open, extensible and cycle-accurate network on chip simulator. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 162–163. IEEE.

[Chen et al., 2011] Chen, J., Li, C., and Gillard, P. (2011). Network-on-chip (noc) topologies and performance: a review. In *Proceedings of the 2011 Newfoundland Electrical and Computer Engineering Conference (NECEC)*, pages 1–6.

[Chiu, 2000] Chiu, G.-M. (2000). The odd-even turn model for adaptive routing. *IEEE Transactions on parallel and distributed systems*, 11(7):729–738. IEEE.

[Chun et al., 2007] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2007). Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204. ACM New York, NY, USA.

[Cong et al., 2014] Cong, L., Wen, W., and Zhiying, W. (2014). A configurable, programmable and software-defined network on chip. In *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*, pages 813–816. IEEE.

[Constantinescu, 2003] Constantinescu, C. (2003). Trends and challenges in vlsi circuit reliability. *IEEE micro*, 23(4):14–19. IEEE.

[Cotret et al., 2016] Cotret, P., Gogniat, G., and Flórez, M. J. S. (2016). Protection of heterogeneous architectures on fpgas: An approach based on hardware firewalls. *Microprocessors and Microsystems*, 42:127–141. Elsevier.

[Cowling et al., 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. of the 7th symposium on Operating systems design and implementation*, pages 177–190.

[Dall'Osso et al., 2012] Dall'Osso, M., Biccari, G., Giovannini, L., Bertozzi, D., and Benini, L. (2012). Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor socs. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 45–48. IEEE.

[Dally and Towles, 2004] Dally, W. J. and Towles, B. P. (2004). *Principles and practices of interconnection networks*. Elsevier.

[Daoud, 2018] Daoud, L. (2018). Secure network-on-chip architectures for mpsoc: Overview and challenges. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 542–543. IEEE.

[Daoud and Rafla, 2018] Daoud, L. and Rafla, N. (2018). Routing aware and runtime detection for infected network-on-chip routers. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 775–778. IEEE.

[Daoud and Rafla, 2019a] Daoud, L. and Rafla, N. (2019a). Analysis of black hole router attack in network-on-chip. In *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 69–72. IEEE.

[Daoud and Rafla, 2019b] Daoud, L. and Rafla, N. (2019b). Detection and prevention protocol for black hole attack in network-on-chip. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, page 22. ACM.

[Diguet et al., 2007] Diguet, J.-P., Evain, S., Vaslin, R., Gogniat, G., and Juin, E. (2007). Noc-centric security of reconfigurable soc. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 223–232. IEEE.

[Dimitrakopoulos et al., 2015] Dimitrakopoulos, G., Psarras, A., and Seitanidis, I. (2015). *Microarchitecture of Network-on-chip Routers*, volume 1025. Springer.

[Dobkin et al., 2009] Dobkin, R. R., Ginosar, R., and Kolodny, A. (2009). Qnoc asynchronous router. *Integration*, 42(2):103–115. Elsevier.

[Dover, 2013] Dover, J. M. (2013). A denial of service attack against the open floodlight sdn controller. *Dover Networks, Tech. Rep.*

[Dumitras et al., 2003] Dumitras, T., Kerner, S., and Marculescu, R. (2003). Towards on-chip fault-tolerant communication. In *Proc. of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003.*, pages 225–232. IEEE.

[Dworkin, 2007] Dworkin, M. J. (2007). *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac.* National Institute of Standards & Technology.

[Eldewahi et al., 2018] Eldewahi, A. E., Hassan, A., Elbadawi, K., and Barry, B. I. (2018). The analysis of mate attack in sdn based on stride model. In *International Conference on Emerging Internetworking, Data & Web Technologies*, pages 901–910. Springer.

[Ellinidou et al., 2018] Ellinidou, S., Sharma, G., Dricot, J.-M., and Markowitch, O. (2018). A SDN solution for system-on-chip world. In *Software Defined Systems (SDS), 2018 Fifth International Conference on*, pages 14–19. IEEE.

[Ellinidou et al., 2019] Ellinidou, S., Sharma, G., Kontogiannis, S., Markowitch, O., Dricot, J.-M., and Gogniat, G. (2019). Microlet: A new sdnoc-based communication protocol for chiplet-based systems. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 61–68. IEEE.

[Ellinidou et al., 2020a] Ellinidou, S., Sharma, G., Markowitch, O., Dricot, J.-M., and Gogniat, G. (2020a). Towards noc protection of ht-greyhole attack. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 309–323. Springer.

[Ellinidou et al., 2020b] Ellinidou, S., Sharma, G., Markowitch, O., Gogniat, G., and Dricot, J.-M. (2020b). A novel network-on-chip security algorithm for tolerating byzantine faults. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE.

[Fan et al., 2012] Fan, G., Yu, H., Chen, L., and Liu, D. (2012). Model based byzantine fault detection technique for cloud computing. In *2012 IEEE Asia-Pacific Services Computing Conference*, pages 249–256. IEEE.

[Fathi and Kia, 2017] Fathi, A. and Kia, K. (2017). A centralized controller as an approach in designing noc. *International Journal of Modern Education and Computer Science*, 9(1):60. Modern Education and Computer Science Press.

[Fawcett, 2006] Fawcett, T. (2006). An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874. Elsevier.

[Fernandes et al., 2016] Fernandes, R., Marcon, C., Cataldo, R., Silveira, J., Sigl, G., and Sepúlveda, J. (2016). A security aware routing approach for noc-based mpsocs. In *Integrated Circuits and Systems Design (SBCCI), 2016 29th Symposium on*, pages 1–6. IEEE.

[Fernandes et al., 2015] Fernandes, R., Oliveira, B., Sepúlveda, J., Marcon, C., and Moraes, F. G. (2015). A non-intrusive and reconfigurable access control to secure nocs. In *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on*, pages 316–319. IEEE.

[Fiorin et al., 2008] Fiorin, L., Palermo, G., Lukovic, S., Catalano, V., and Silvano, C. (2008). Secure memory accesses on networks-on-chip. *IEEE Transactions on Computers*, 57(9):1216–1229. IEEE.

[Fiorin et al., 2007] Fiorin, L., Silvano, C., and Sami, M. (2007). Security aspects in networks-on-chips: Overview and proposals for secure implementations. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 539–542. IEEE.

[Flynn, 1997] Flynn, D. (1997). Amba: enabling reusable on-chip designs. *IEEE micro*, 17(4):20–27. IEEE.

[Force, 2005] Force, T. (2005). High performance microchip supply. *Annual Report. Defense Technical Information Center (DTIC), USA*.

[Foundation, 2015] Foundation, O. N. (2015). Openflow switch specification version 1.5. 1 (protocol version 0x06).

[Frank, 2010] Frank, A. (2010). Uci machine learning repository. irvine, ca: University of california, school of information and computer science. *http://archive. ics. uci. edu/ml*.

[Frey and Yu, 2015] Frey, J. and Yu, Q. (2015). Exploiting state obfuscation to detect hardware trojans in noc network interfaces. In *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4. IEEE.

[Glass and Ni, 1992] Glass, C. J. and Ni, L. M. (1992). The turn model for adaptive routing. *ACM SIGARCH Computer Architecture News*, 20(2):278–287. ACM.

[Goossens et al., 2005] Goossens, K., Dielissen, J., and Radulescu, A. (2005). Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421. IEEE.

[Gorman, 2012] Gorman, C. (2012). Counterfeit chips on the rise. *IEEE Spectrum*, 49(6). IEEE.

[Grammatikakis et al., 2014] Grammatikakis, M. D., Papadimitriou, K., Petrakis, P., Papagrigoriou, A., Kornaros, G., Christoforakis, I., and Coppola, M. (2014). Security effectiveness and a hardware firewall for mpsocs. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pages 1032–1039. IEEE.

[Grecu et al., 2008] Grecu, C., Ivanov, A., Saleh, R., Rusu, C., Anghel, L., Pande, P. P., and Nuca, V. (2008). A flexible network-on-chip simulator for early design space exploration. In *2008 1st Microsystems and Nanoelectronics Research Conference*, pages 33–36. IEEE.

[Hegedûs et al., 2005] Hegedûs, A., Maggio, G. M., and Kocarev, L. (2005). A ns-2 simulator utilizing chaotic maps for network-on-chip traffic analysis. In *2005 IEEE International Symposium on Circuits and Systems*, pages 3375–3378. IEEE.

[Hernan et al., 2006] Hernan, S., Lambert, S., Ostwald, T., and Shostack, A. (2006). Threat modeling-uncover security design flaws using the stride approach. *MSDN Magazine-Louisville*, pages 68–75. San Francisco, CA: CMP Media Inc., c2000-.

[Hilbrich and van Kampenhout, 2010] Hilbrich, R. and van Kampenhout, R. (2010). Dynamic reconfiguration in noc-based mpsocs in the avionics domain. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, pages 56–57.

[Hong et al., 2015] Hong, S., Xu, L., Wang, H., and Gu, G. (2015). Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, volume 15, pages 8–11.

[Hoskote et al., 2007] Hoskote, Y., Vangal, S., Singh, A., Borkar, N., and Borkar, S. (2007). A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61. IEEE.

[Howard et al., 2010] Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., et al. (2010). A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 108–109. IEEE.

[Hu et al., 2014] Hu, F., Hao, Q., and Bao, K. (2014). A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206. IEEE.

[Hu and Marculescu, 2004] Hu, J. and Marculescu, R. (2004). Dyad: smart routing for networks-on-chip. In *Proceedings of the 41st annual Design Automation Conference*, pages 260–263. ACM.

[Hu et al., 2015] Hu, Z., Wang, M., Yan, X., Yin, Y., and Luo, Z. (2015). A comprehensive security architecture for sdn. In *2015 18th International Conference on Intelligence in Next Generation Networks*, pages 30–37. IEEE.

[Hubner et al., 2005] Hubner, M., Paulsson, K., and Becker, J. (2005). Parallel and flexible multiprocessor system-on-chip for adaptive automotive applications based on xilinx microblaze soft-cores. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 6–pp. IEEE.

[Hussain and Guo, 2017] Hussain, M. and Guo, H. (2017). Packet leak detection on hardware-trojan infected nocs for mpsoc systems. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 85–90. ACM.

[Intel, 2017] Intel (2017). "New Intel Core Processor Combines Highperformance CPU with Custom Discrete Graphics From AMD to Enable Sleeker, Thinner Devices,".

[Issariyakul and Hossain, 2009] Issariyakul, T. and Hossain, E. (2009). Introduction to network simulator 2 (ns2). In *Introduction to network simulator NS2*, pages 1–18. Springer.

[Iyer, 2016] Iyer, S. S. (2016). Heterogeneous integration for performance and scaling. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 6(7):973–982. IEEE.

[Jain et al., 2007] Jain, L., Al-Hashimi, B., Gaur, M., Laxmi, V., and Narayanan, A. (2007). Nirgam: a simulator for noc interconnect routing and application modeling. In *Design, automation and test in Europe conference*, pages 16–20. IEEE.

[Jerger et al., 2017] Jerger, N. E., Krishna, T., and Peh, L.-S. (2017). On-chip networks. *Synthesis Lectures on Computer Architecture*, 12(3):1–210. Morgan & Claypool Publishers.

[Jiang et al., 2013] Jiang, N., Becker, D. U., Michelogiannakis, G., Balfour, J., Towles, B., Shaw, D. E., Kim, J., and Dally, W. J. (2013). A detailed and flexible cycle-accurate network-on-chip simulator. In *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 86–96. IEEE.

[JS et al., 2015] JS, R., Ancajas, D. M., Chakraborty, K., and Roy, S. (2015). Runtime detection of a bandwidth denial attack from a rogue network-on-chip. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, page 8. ACM.

[Kannan et al., 2015] Kannan, A., Jerger, N. E., and Loh, G. H. (2015). Enabling interposer-based disintegration of multi-core processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–558. IEEE.

[Kermani and Kleinrock, 1979] Kermani, P. and Kleinrock, L. (1979). Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, 3(4):267–286. Elsevier.

[Kim et al., 2005] Kim, J., Park, D., Theocharides, T., Vijaykrishnan, N., and Das, C. R. (2005). A low latency router supporting adaptivity for on-chip interconnects. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 559–564. IEEE.

[Klöti et al., 2013] Klöti, R., Kotronis, V., and Smith, P. (2013). Openflow: A security analysis. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE.

[Kocher et al., 2019] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2019). Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE.

[Kotla et al., 2010] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2010). Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27(4):1–39. ACM New York, NY, USA.

[Krishnamurthy et al., 2014] Krishnamurthy, A., Chandrabose, S. P., and Gember-Jacobson, A. (2014). Pratyaastha: an efficient elastic distributed

sdn control plane. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 133–138.

[Kundu and Chattopadhyay, 2018] Kundu, S. and Chattopadhyay, S. (2018). *Network-on-chip: the next generation of system-on-chip integration*. CRC press.

[LAMPORT et al., 1982] LAMPORT, L., SHOSTAK, R., and PEASE, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Lantz et al., 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6.

[Li et al., 2016] Li, W., Meng, W., and Kwok, L. F. (2016). A survey on openflow-based software defined networks: Security challenges and countermeasures. *Journal of Network and Computer Applications*, 68:126–139. Elsevier.

[Liang et al., 2000] Liang, J., Swaminathan, S., and Tessier, R. (2000). asoc: A scalable, single-chip communications architecture. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 37–46. IEEE.

[Lipp et al., 2018] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown. *arXiv preprint arXiv:1801.01207*.

[Lu et al., 2005] Lu, Z., Thid, R., Millberg, M., Nilsson, E., and Jantsch, A. (2005). Nnse: Nostrum network-on-chip simulation environment. In *Swedish system-on-chip conference*.

[Lynn, 2006] Lynn, B. (2006). Pbc library manual 0.5. 11.

[Ma et al., 2014] Ma, S., Huang, L., Lai, M., and Shi, W. (2014). *Networks-on-chip: From Implementations to Programming Paradigms*. Morgan Kaufmann.

[Malekpour et al., 2017] Malekpour, A., Ragel, R., Ignjatovic, A., and Parameswaran, S. (2017). Trojanguard: Simple and effective hardware trojan mitigation techniques for pipelined mpsocs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 19. ACM.

[Martins and Guyennet, 2010] Martins, D. and Guyennet, H. (2010). Wireless sensor network attacks and security mechanisms: A short survey. In *2010 13th International Conference on Network-Based Information Systems*, pages 313–320. IEEE.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74. ACM.

[Millberg et al., 2004a] Millberg, M., Nilsson, E., Thid, R., and Jantsch, A. (2004a). 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. IEEE.

[Millberg et al., 2004b] Millberg, M., Nilsson, E., Thid, R., Kumar, S., and Jantsch, A. (2004b). The nostrum backbone-a communication protocol stack for networks on chip. In *17th International Conference on VLSI Design. Proceedings.*, pages 693–696. IEEE.

[Miraz et al., 2015] Miraz, M. H., Ali, M., Excell, P. S., and Picking, R. (2015). A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont). In *2015 Internet Technologies and Applications (ITA)*, pages 219–224. IEEE.

[Miyaji et al., 2001] Miyaji, A., Nakabayashi, M., and Takano, S. (2001). New explicit conditions of elliptic curve traces for FR-reduction. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 84(5):1234–1243. The Institute of Electronics, Information and Communication Engineers.

[Moore, 1998] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85. IEEE.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[Niehaus et al., 1989] Niehaus, J. A., Fleck, R. G., Li, S., and Strong, B. D. (1989). Digital crossbar switch. US Patent 4,852,083.

[Nilsson, 2002] Nilsson, E. (2002). Design and implementation of a hot-potato switch in a network on chip. *Mémoire, Departement of Microelectronics and Information Technology, Royal Institute of Technology.* Citeseer.

[NVIDIA, 2016] NVIDIA, T. (2016). P100 white paper. *NVIDIA Corporation*.

[Ogras and Marculescu, 2005] Ogras, U. Y. and Marculescu, R. (2005). Application-specific network-on-chip architecture customization via long-range link insertion. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 246–253. IEEE.

[Owens et al., 2007] Owens, J. D., Dally, W. J., Ho, R., Jayasimha, D., Keckler, S. W., and Peh, L.-S. (2007). Research challenges for on-chip interconnection networks. *IEEE micro*, 27(5):96–108. IEEE.

[Panda and Khilar, 2015] Panda, M. and Khilar, P. M. (2015). Distributed byzantine fault detection technique in wireless sensor networks based on hypothesis testing. *Computers & Electrical Engineering*, 48:270–285. Elsevier.

[Pfaff et al., 2015] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. (2015). The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 117–130. USENIX Association.

[Phemius et al., 2014] Phemius, K., Bouet, M., and Leguay, J. (2014). Disco: Distributed multi-domain sdn controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE.

[Philip et al., 2014] Philip, J., Kumar, S., Norige, E., Hassan, M., and Mitra, S. (2014). Asymmetric mesh noc topologies. US Patent 8,819,616.

[Rajesh et al., 2018] Rajesh, J., Chakraborty, K., and Roy, S. (2018). Hardware trojan attacks in soc and noc. In *The Hardware Trojan War*, pages 55–74. Springer.

[Rajsuman, 2000] Rajsuman, R. (2000). *System-on-a-chip: Design and Test*. Artech House, Inc.

[Rantala et al., 2006] Rantala, V., Lehtonen, T., Plosila, J., et al. (2006). *Network on chip routing algorithms*. Citeseer.

[Rijpkema et al., 2003] Rijpkema, E., Goossens, K., Rădulescu, A., Dielissen, J., van Meerbergen, J., Wielage, P., and Waterlander, E. (2003). Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proceedings-Computers and Digital Techniques*, 150(5):294–302. IET.

[Rong and Liu, 2017] Rong, R. and Liu, J. (2017). Distributed mininet with symbiosis. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–6. IEEE.

[Rosenblatt, 1956] Rosenblatt, M. (1956). A central limit theorem and a strong mixing condition. *Proceedings of the National Academy of Sciences of the United States of America*, 42(1):43. National Academy of Sciences.

[Ruaro et al., 2020] Ruaro, M., Caimi, L. L., and Moraes, F. G. (2020). A systemic and secure sdn framework for noc-based many-cores. *IEEE Access*, 8:105997–106008. IEEE.

[Ruaro et al., 2018] Ruaro, M., Medina, H. M., Amory, A. M., and Moraes, F. G. (2018). Software-defined networking architecture for noc-based many-cores. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

[Ruaro et al., 2017] Ruaro, M., Medina, H. M., and Moraes, F. G. (2017). Sdn-based circuit-switching for many-cores. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 385–390. IEEE.

[Ruaro et al., 2019] Ruaro, M., Velloso, N., Jantsch, A., and Moraes, F. G. (2019). Distributed sdn architecture for noc-based many-core socs. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, pages 1–8. ACM.

[Salvador et al., 2017] Salvador, I.-D., Remberto, S.-A., Brox, M., and Ortiz, M. A. (2017). Software defined network controller: A neat solution administration for reconfigurable multi-core noc. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–4. IEEE.

[Sandoval-Arechiga et al., 2016] Sandoval-Arechiga, R., Parra-Michel, R., Vazquez-Avila, J., Flores-Troncoso, J., and Ibarra-Delgado, S. (2016). Software defined networks-on-chip for multi/many-core systems: A performance evaluation. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, pages 129–130. ACM.

[Sandoval-Arechiga et al., 2015] Sandoval-Arechiga, R., Vazquez-Avila, J., Parra-Michel, R., Flores-Troncoso, J., and Ibarra-Delgado, S. (2015). Shifting the network-on-chip paradigm towards a software defined network architecture. In *Computational Science and Computational Intelligence (CSCI), 2015 International Conference on*, pages 869–870. IEEE.

[Scheffe, 1999] Scheffe, H. (1999). *The analysis of variance*, volume 72. John Wiley & Sons.

[Schmidt et al., 1993] Schmidt, O. S., Husted, R. R., Van Sickle, W., Dauterman, T. L., and Rohn, D. R. (1993). Processor for a programmable controller. US Patent 5,265,005.

[Scionti et al., 2016] Scionti, A., Mazumdar, S., and Portero, A. (2016). Software defined network-on-chip for scalable cmps. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 112–115. IEEE.

[Scionti et al., 2018] Scionti, A., Mazumdar, S., and Portero, A. (2018). Towards a scalable software defined network-on-chip for next generation cloud. *Sensors*, 18(7):2330. Multidisciplinary Digital Publishing Institute.

[Seemuth et al., 2015] Seemuth, D. P., Davoodi, A., and Morrow, K. (2015). Automatic die placement and flexible i/o assignment in 2.5 d ic design. In *Sixteenth International Symposium on Quality Electronic Design*, pages 524–527. IEEE.

[Seiculescu et al., 2010] Seiculescu, C., Murali, S., Benini, L., and De Micheli, G. (2010). A method to remove deadlocks in networks-on-chips with wormhole flow control. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1625–1628. IEEE.

[Sepulveda et al., 2017a] Sepulveda, J., Fernandes, R., Marcon, C., Florez, D., and Sigl, G. (2017a). A security-aware routing implementation for dynamic data protection in zone-based mpsoc. In *Integrated Circuits and Systems Design (SBCCI), 2017 30th Symposium on*, pages 59–64. IEEE.

[Sepulveda et al., 2016] Sepulveda, J., Flórez, D., Fernandes, R., Marcon, C., Gogniat, G., and Sigl, G. (2016). Towards risk aware nocs for data protection in mpsocs. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2016 11th International Symposium on*, pages 1–8. IEEE.

[Sepulveda et al., 2017b] Sepulveda, J., Flórez, D., Immler, V., Gogniat, G., and Sigl, G. (2017b). Efficient security zones implementation through hierarchical group key management at noc-based mpsocs. *Microprocessors and Microsystems*, 50:164–174. Elsevier.

[Sepulveda et al., 2014] Sepulveda, J., Gogniat, G., Flórez, D., Diguet, J.-P., Zeferino, C., and Strum, M. (2014). Elastic security zones for noc-based 3d-mpsocs. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 506–509. IEEE.

[Sepúlveda et al., 2017] Sepúlveda, J., Zankl, A., Flórez, D., and Sigl, G. (2017). Towards protected mpsoc communication for information protection against a malicious noc. *Procedia computer science*, 108:1103–1112. Elsevier.

[Sethi and Sarangi, 2017] Sethi, P. and Sarangi, S. R. (2017). Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, 2017. Hindawi.

[Sezer et al., 2013] Sezer, S., Scott-Hayward, S., Chouhan, P. K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., and Rao, N. (2013). Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43. IEEE.

[Sharma et al., 2018] Sharma, G., Ellinidou, S., Kuchta, V., Sahu, R. A., Markowitch, O., and Dricot, J.-M. (2018). Secure communication on noc based mpsoc. In *International Conference on Security and Privacy in Communication Systems*, pages 417–428. Springer.

[Sharma et al., 2019] Sharma, G., Kuchta, V., Anand Sahu, R., Ellinidou, S., Bala, S., Markowitch, O., and Dricot, J.-M. (2019). A twofold group key agreement protocol for noc-based mpsocs. *Transactions on Emerging Telecommunications Technologies*, 30(6):e3633. Wiley Online Library.

[Sharma et al., 2017] Sharma, G., Sahu, R. A., Kuchta, V., Markowitch, O., and Bala, S. (2017). Authenticated Group Key Agreement Protocol Without Pairing. In *International Conference on Information and Communications Security*, pages 606–618. Springer.

[Silva et al., 2019] Silva, R. S., Cruz, P. P., Kreutz, M. E., and Pereira, M. M. (2019). Communication latency evaluation on a software-defined network-on-chip. In *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–7. IEEE.

[Smeets, 2018] Smeets, M. (2018). A matter of time: On the transitory nature of cyberweapons. *Journal of Strategic Studies*, 41(1-2):6–32. Taylor & Francis.

[Sokolova and Lapalme, 2009] Sokolova, M. and Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437. Elsevier.

[Soultana Ellinidou, 2019] Soultana Ellinidou, Gaurav Sharma, T. R. T. V. O. M. J.-M. D. (2019). "sspsoc: A secure sdn-based protocol over mpsoc.". volume 2019. Hindawi.

[Sutardja, 2015] Sutardja, S. (2015). 1.2 the future of ic design innovation. In *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, pages 1–6. IEEE.

[Syverson, 1994] Syverson, P. (1994). A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187–191. IEEE.

[Teng and Wu, 2016] Teng, J. and Wu, C. (2016). An identity-based group key agreement protocol for low-power mobile devices. *Chinese Journal of Electronics*, 25(4):726–733. IET.

[Tomonori, 2013] Tomonori, F. (2013). Introduction to ryu sdn framework. *Open Networking Summit*, pages 1–14.

[Tripathi et al., 2013] Tripathi, M., Gaur, M. S., and Laxmi, V. (2013). Comparing the impact of black hole and gray hole attack on leach in wsn. *Procedia Computer Science*, 19:1101–1107. Elsevier.

[Van Bulck et al., 2018] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association.*

[Varga, 2010] Varga, A. (2010). Omnet++. In *Modeling and tools for network simulation*, pages 35–59. Springer.

[Vijayaraghavan et al., 2017] Vijayaraghavan, T., Eckert, Y., Loh, G. H., Schulte, M. J., Ignatowski, M., Beckmann, B. M., Brantley, W. C., Greathouse, J. L., Huang, W., Karunanithi, A., et al. (2017). Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96. IEEE.

[Wentzlaff et al., 2007] Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown III, J. F., and Agarwal, A. (2007). On-chip interconnection architecture of the tile processor. *IEEE micro*, 27(5):15–31. IEEE.

[Wu et al., 2016] Wu, S.-Y., Lin, C., Chiang, M., Liaw, J., Cheng, J., Yang, S., Tsai, C., Chen, P., Miyashita, T., Chang, C., et al. (2016). A 7nm cmos platform technology featuring 4 th generation finfet transistors with a 0.027 um 2 high density 6-t sram cell for mobile soc applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2–6. IEEE.

[Xie et al., 2018] Xie, J., Yu, F. R., Huang, T., Xie, R., Liu, J., Wang, C., and Liu, Y. (2018). A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, 21(1):393–430. IEEE.

[Xu et al., 2015] Xu, J., Wang, K., Wang, C., Hu, F., Zhang, Z., Xu, S., and Wu, J. (2015). Byzantine fault-tolerant routing for large-scale wireless sensor networks based on fast ecdsa. *Tsinghua Science and Technology*, 20(6):627–633. TUP.

[Yan and Yu, 2015] Yan, Q. and Yu, F. R. (2015). Distributed denial of service attacks in software-defined networking with cloud computing. *IEEE Communications Magazine*, 53(4):52–59. IEEE.

[Yan et al., 2016] Yan, Q., Yu, F. R., Gong, Q., and Li, J. (2016). Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges. *IEEE Communications Surveys & Tutorials*, 18(1):602–622. IEEE.

[Zhang et al., 2018a] Zhang, H., Cai, Z., Liu, Q., Xiao, Q., Li, Y., and Cheang, C. F. (2018a). A survey on security-aware measurement in sdn. *Security and Communication Networks*, 2018. Hindawi.

[Zhang et al., 2018b] Zhang, L., Wang, X., Jiang, Y., Yang, M., Mak, T., and Singh, A. K. (2018b). Effectiveness of ht-assisted sinkhole and black-hole denial of service attacks targeting mesh networks-on-chip. *Journal of Systems Architecture*, 89:84–94. Elsevier.

[Zhang et al., 2011] Zhang, Y., Zheng, Z., and Lyu, M. R. (2011). Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 444–451. IEEE.

[Zhou and Zhu, 2017] Zhou, X. and Zhu, Z. (2017). A dynamic task mapping algorithm for sdnoc. *Microelectronics Journal*, 63:58–65. Elsevier.