Multi-Level Fault-Tolerance in Networks-on-Chip

C. Rusu

UNIVERSITE DE GRENOBLE
INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque **978-2-84813-158-0**

T H E S E

pour obtenir le grade de
DOCTEUR DE L'Université de Grenoble
délivré par l'Institut polytechnique de Grenoble

Spécialité : Micro et Nano Electronique

préparée au laboratoire TIMA

dans le cadre de **l'Ecole Doctorale d'Électronique, Électrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement

par
Claudia RUSU

le 10 septembre 2010

Multi-Level Fault-Tolerance in Networks-on-Chip

DIRECTEUR DE THESE : Michael NICOLAIDIS
CO-DIRECTEUR DE THESE : Lorena ANGHEL

JURY

M.	Frédéric ROUSSEAU	, Président
M.	Ian O'CONNOR	, Rapporteur
M.	Jacques COLLET	, Rapporteur
M.	Michael NICOLAIDIS	, Directeur de thèse
Mme.	Lorena ANGHEL	, Co-encadrant
M.	Kees GOOSSENS	, Examinateur
M.	Marcello COPPOLA	, Invité
M.	Sorin COTOFANA	, Invité

Acknowledgements

First, I would like to thank my PhD director, Dr. Michael Nicolaidis, head of ARIS group of TIMA, for having accepted to supervise my thesis.

I express my gratitude to Prof. Lorena Anghel, co-director of my thesis, who, from the very beginning, believed in my project and fully supported me all over these years. Thanks for the time dedicated to my work supervision and for the excellent advices.

I would like to thank Prof. Frédéric Rousseau, from TIMA, SLS group, for presiding my thesis jury, as well as for his useful advices and the professionalism he showed during our teaching collaboration at Polytech'Grenoble. Thanks to Prof. Ian O'Connor from INL Lyon and Dr. Jacques Collet from LAAS Toulouse for their hard work of revising my thesis manuscript and for their suggestions to improve it. Thanks to Prof. Kees Goossens from TU Eindhoven for his relevant questions and comments that contributed to the completeness of the manuscript and the extension of my work. I also thank Dr. Marcello Coppola from ST Grenoble and Prof. Sorin Cotofana from TU Delft for their participation to my PhD defense.

I want to thank Dr. Cristian Grecu from SoC Laboratory, University of British Columbia (now with MIT) for his close collaboration in our research regarding the checkpoint, for his ideas and good perspective orientation of our work. Thanks to Prof. Dimiter Avresky from IRIANC Boston for his collaboration during our research in 3D NoC routing, for his scientific rigor, good organization of the work and practical advices for my defense. I also want to acknowledge Vladimir Pasca, my colleague from TIMA, for the productive discussions we had and for his real help with the implementation and simulations of fault-tolerant mechanisms for links. I want to thank also my other research collaborators, from whom I learnt a lot of new things, Dr. Dan Alexandrescu from iRoC and Antonin Bougerol from EADS Paris.

I thank my colleagues Gilles Bizot, Yu Hai, Diarga Fall and Seddik Benhammadi, who supported and helped me during these years. Many thanks to my colleagues Gilles Bizot, Fabien Chaix and Thierry Bonnoît, as well as to Prof. Frédéric Rousseau, for their great help in correcting and improving the French summary of my thesis.

I want to thank Prof. Dominique Borrione, head of TIMA Laboratory, for her professional advices and support. Thanks also to Prof. Eric Gascard, my three year monitor tutor at Polytech'Grenoble, for his assistance in my professional development. Thanks to Prof. Nacer-Eddine Zergainoh for his good advices. I want also to acknowledge Prof. Pierre Gentil, head of EEATS doctoral school, for his substantial help at the very beginning of my stay in Grenoble.

Finally, I thank my family and friends for their support, and especially my husband, Marius, without whom nothing of these would have been possible.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Nowadays, the information processing is more and more moving away from just personal computers to embedded systems, as the key goals are to make information available anytime and anywhere and to build ambient intelligence into our environment. Embedded systems have applications in all domains and concern both every day life and critical missions: audio, video, home electronics, automotive electronics, avionics, telecommunication, medical systems, authentication systems, cryptography, industrial production systems, robotics etc.

Systems-on-chip (*SoC*) are embedded systems that comprise several components into a single integrated circuit (IC): processors, memories, peripherals, power management circuits and others. A SoC with more than one processor core is called *multiprocessor systems on chip* (*MP-SoC*).

The development of mobile telephony, personal digital assistants (PDAs), and multimedia technologies in general created new needs: video decoding, interactive three-dimensional games, digital audio decoding etc. These applications require more and more computational power [ITR09a], as depicted in Fig. 1-1.

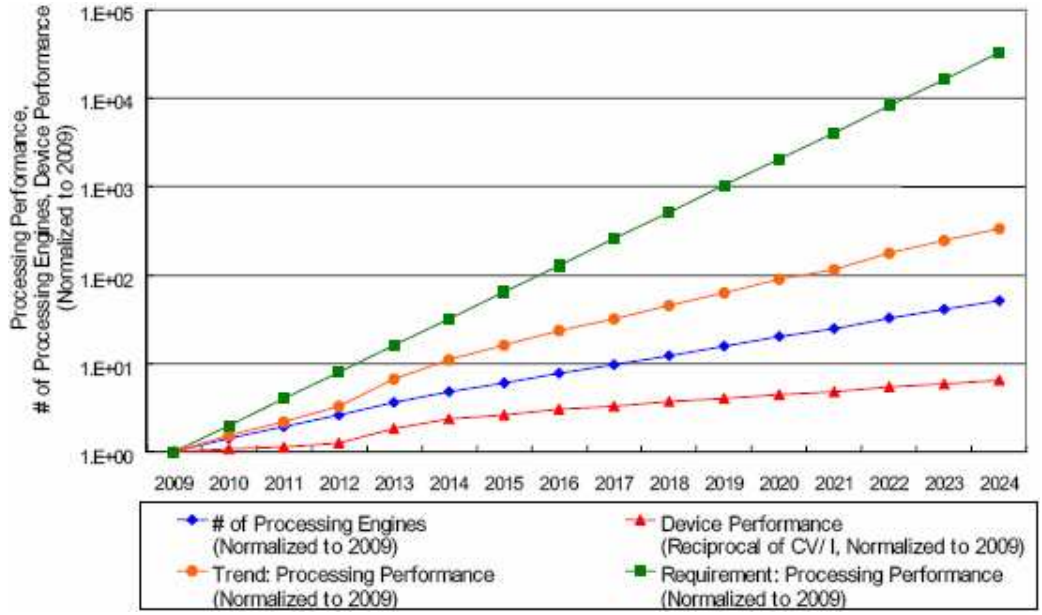


Fig. 1-1 SoC consumer portable processing performance trends (ITRS 2009)

Integrating many processors in a single chip is a potential solution to cope with the continuously demanding computing power of the embedded applications and the aggressive time-to-market constraint. The number of *processing engines* (*PEs*) is projected to significantly increase in the following years [ITR09a] (Fig. 1-2) and reach "thousand cores" by 2020.

Face to the huge complexity, specific constraints of embedded systems must be taken into account, such as low energy consumption, small silicon area or manufacturability and yield.

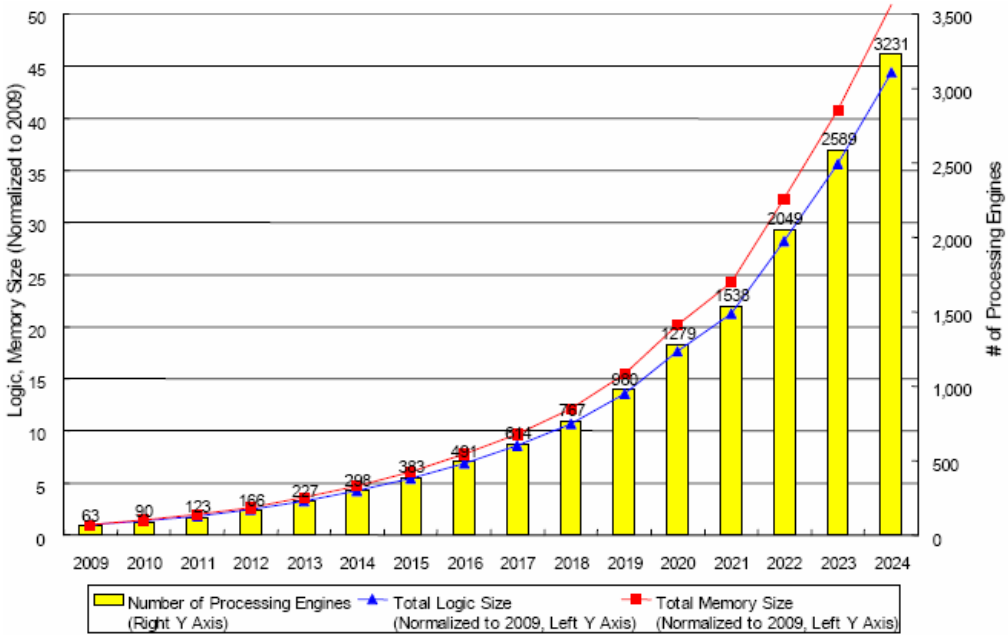


Fig. 1-2 SoC consumer portable design complexity trends (ITRS 2009)

On the technology side, following *Moore's law*, higher and higher degree of integration is expected [Int05]. While transistors continue to improve in performance at smaller size, the wiring connecting them presents increasing delays [ITR05] (Fig. 1-3). Repeaters can be added, to mitigate the delay, but they consume power and area. Today, more than 50% of dynamic power consumption is due to interconnects and this rate is projected to increase [MKWS04] [JLV05]. Therefore, meeting power objectives is becoming as important as meeting performance targets [KNM04], if not more.

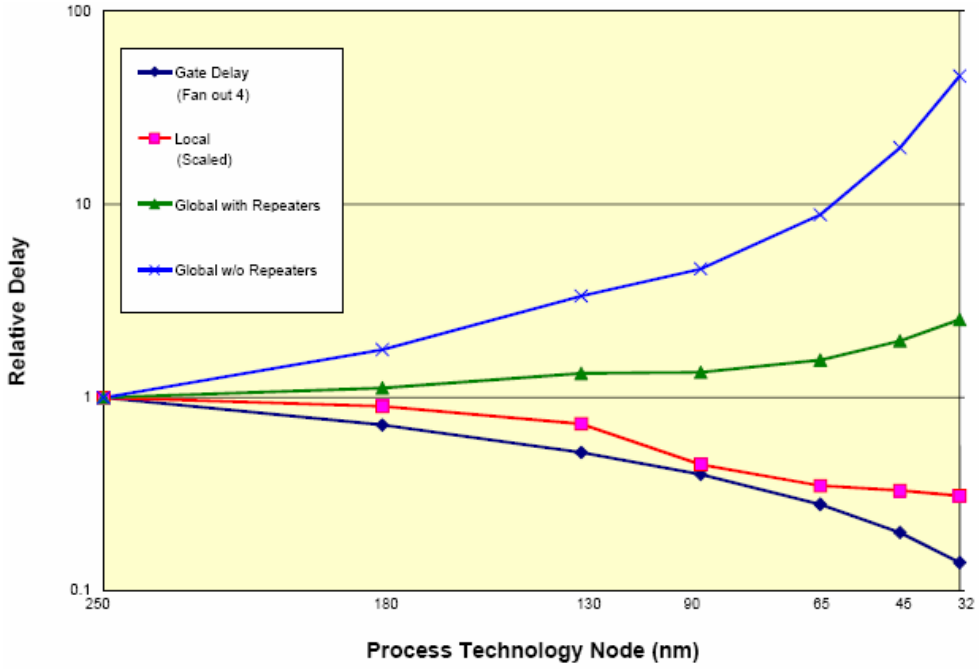


Fig. 1-3 Relative delay for local and global wires and for logic gates (ITRS 2005)

3D integration, intensively studied in the last few years, offers *more Moore perspective* for ICs, especially for SoCs dedicated to multimedia and mobile applications. 3D

integration consists of stacking integrated circuits and connecting them vertically by using *through-silicon-vias* (*TSVs*). Long horizontal interconnects are replaced with shorter vertical ones, thus reducing global wire delays and overall dynamic power consumption. Another great advantage of 3D integration by stacked layers is that beside of the homogeneous integration, the integration of heterogeneous devices and technologies becomes possible.

However, the lack of design tools that handle floorplanning, place and route, but also TSVs and 3D die stacking currently delays the mainstream acceptance of this technology. Besides the restrictions related to the heterogeneity irregularities, restrictions related to the TSV count, size and positions impact the stack vertical connectivity [DM09]. Moreover, 3D integration brings significant yield problems, mainly due to the additional manufacturing processes such as wafer thinning and bonding.

With the continuous increasing of MP-SoC complexity, as more and more cores are integrated, the communication among them becomes a scalability and time-to-market bottleneck.

Networks-on-chip

The *network-on-chip* (*NoC*) design paradigm addresses the ever increasing complexity, delay and power dissipation of global 2D on-chip interconnects by offering structured interconnect fabrics consisting of *wire segments* (*links*) and *routing blocks* (*routers* or *switches*), as depicted in Fig. 1-4.c. The basic traditional communication systems are the *point-to-point* (*P2P*) and the busses, depicted in Fig. 1-4.a and b. The NoC communication architecture is intended to gather the advantages of these main existing architectures (scalability, low communication latency), while reducing their drawbacks (such as the number of long wires in P2P and the communication latency in buses).

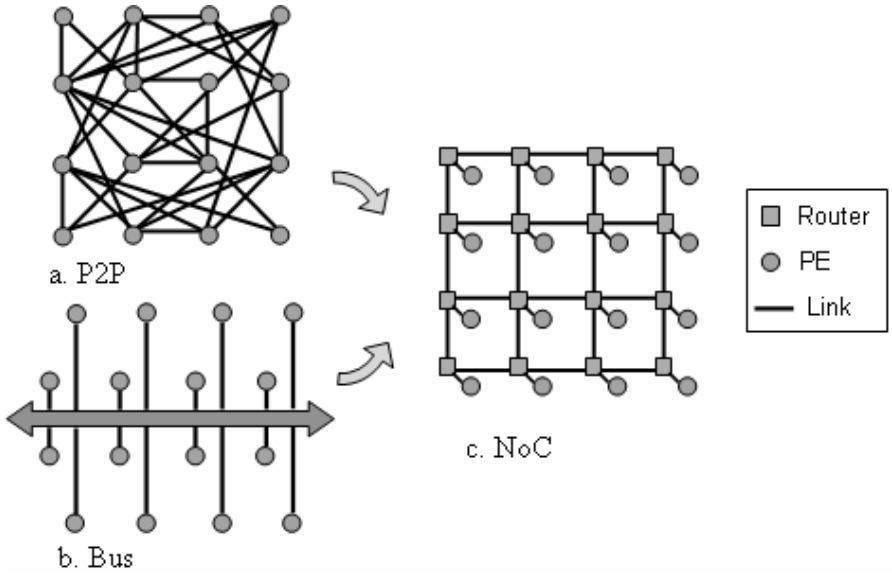


Fig. 1-4 NoC vs. traditional communication systems

The NoC application is seen as a set of tasks running on individual processing elements (PEs) and communicating through the NoC. The communication protocol stack is similar to that proposed by ISO-OSI [Zim88] and presented in Fig. 1-5.

Fig. 1-5 Communication protocol stack

For typical SoC applications, the nature of the communication pattern is mostly heterogeneous, with certain task sets exchanging data more frequently than others. Besides, some tasks can be critical or real-time. Different applications with their own disjoint task sets can also run simultaneously on the NoC-based system [HCG07]. Even if the tasks in different disjoint sets do not exchange data with each other, they compete for NoC resources (routers, channels). Mapping the application tasks to PEs is an important step in the NoC design flow, and is generally performed with the objective of minimizing communication latency and power dissipation [MCM+05].

NoCs were firstly proposed for 2D chips, but with the emerging 3D integration paradigm, NoCs can be extended to 3D topologies, given their modularity and scalability.

Yield and Reliability

With the technology downscaling, SoCs are more prone to different factors that dramatically impact their yield and reliability. Catastrophic defects may occur in the design, due to manufacturing defects. Besides, the variability impact is stronger for smaller feature sizes [ITR09b]. Thus, parametric faults related to device and interconnect variability exhibit faulty behavior similar with that induced by catastrophic defects. The sources of such failures are: process variations, lifetime variations and intrinsic noise. The failure rate of three commonly used CMOS circuits (SRAM cell, latch and inverter) for advance technologies, as presented in [ITR09c], is depicted in Fig. 1-6.

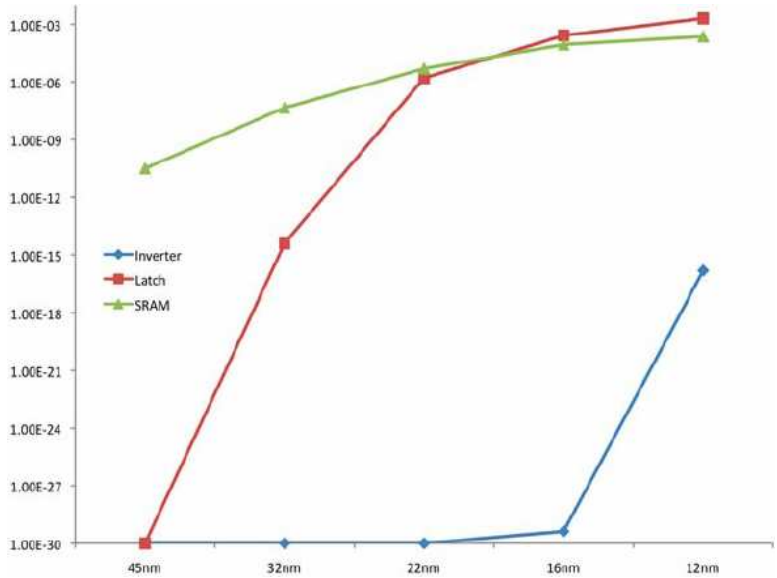


Fig. 1-6 Variability-induced failure rates for three canonical circuit types (ITRS 2009)

Since it is common to have in current SoCs many millions of SRAM bits, millions of latches and inverters, variability leads to greater parametric yield loss with respect to both noise margins and leakage power.

With the feature size diminishing, the embedded memory *soft-error rate* (*SER*) increases. In the mean time, soft errors impact field-level product reliability, not only for embedded memories, but also for logic and latches as well [ITR09c].

Undetected or untreated faults and errors produced at physical level can propagate at higher levels of the communication protocol stack and can impact the correct operation of different components of the system (routers and links, for NoCs), or, eventually, can lead to the failure of the entire system.

Since designs are too large for a cost-effective functional test after manufacturing, adding fault-tolerant mechanisms into the design is mandatory, especially for highly reliable applications, such as in automotive and avionics sectors, but not only. Future many cores platforms will be affected by high defect rate and the same problem will be applied to them, as specified by ITRS roadmap : "Relaxing the requirement of 100% correctness for devices and interconnects may dramatically reduce costs of manufacturing, verification, and test. Such a paradigm shift will likely be forced in any case by technology scaling, which leads to more transient and permanent failures of signals, logic values, devices, and interconnects. (…) Potential solutions include automatic introduction of redundant logic and on-chip reconfigurability for fault tolerance, development of adaptive and self-correcting or self-healing circuits, and software-based fault-tolerance" [ITR09c].

***

Both application specifics and restrictions that apply to SoCs (such as limited power consumption, smaller area and weight) must be taken into account when designing fault-tolerant mechanisms. In this thesis we will discuss fault-tolerant solutions for NoCs, at different levels of the communication protocol stack.

The rest of the thesis is organized as follows:

Chapter 2 presents the context and motivation for this work and defines several classes of problems that will be addressed in this work. Therefore we address different levels of the communication protocol stack: application, transport, routing, and data link levels. Several open questions are formulated, which will be treated in the following chapters.

Chapter 3 analyzes the related works. We conclude that there are no existing works handling well all these questions regarding the fault-tolerance of 2D and 3D NoCs.

Chapter 4 proposes a model for assessing the checkpoint and rollback recovery impact on the NoC performance. Methods to improve the checkpointing scalability are also proposed, considering the application specifics.

Chapter 5 proposes a reconfigurable inter-layer routing methodology for 3D NoCs with any topologies of 2D layers and many irregularities of vertical link distribution. Efficient routing algorithms specifically designed for 2D layers are reused. Adaptability to application specifics is also considered.

Chapter 6 proposes an adaptable fault-tolerant mechanism for data link and transport level, considering the application characteristics. This mechanism can be applied both to 2D links and vertical links in 3D NoCs.

Chapter 7 concludes this thesis and provides perspectives.

Chapter 2. Dependability of NoC-based Systems

This chapter presents the context and motivation of this thesis. With the increasing complexity of nowadays systems-on-chip and of their communication requests, networks-on-chip (NoCs) are generally viewed as the ultimate solution for designing modular and scalable communication architectures. However, even if chips benefit from continuous technology downscaling, their area, power consumption and communication latency keep on increasing. The 3D integration paradigm seems to be a promising solution to cope with very large chips problems. However, technology miniaturization and 3D integration, which is not mature yet, make 3D SoCs prone to several types of faults, which may lead to errors and can in turn provoke failures of the entire application, if not addressed properly.

2.1 System-on-Chip: Complexity, Efficiency and Dependability

Systems-on-chip are embedded systems that integrate all components of the system on the same chip: processors, memories, etc. Face to the new demands for SoCs (more *functionality*, more *complexity*), the time spent in research, development, design and test of new products increases, under the pressure of shorter time-to-market. Platform-based design, multi-application support, regular and *reconfigurable* architectures and network-on-chip (*NoC*) for communication are some of the approaches proposed recently to deal with SoC complexity. 3D integration has recently emerged as a potential solution to deal with the problems of large 2D chips.

Besides their increased functionality, SoCs must also be *efficient* and *dependable*.

Efficiency includes several aspects: *energy*, *weight*, *code-size*, *run-time*, and *cost*. Many SoCs are mobile systems, powered by batteries. As a consequence, their weight and dimensions must be small. Customers expect long run-times from batteries, but battery technology improves at a slower rate than SoC functionality increases. Therefore, compact and power-aware architectures must be designed to deal with these requests. Typically, as there are no hard disks or huge memories to store embedded software, the code must be small. With least amount of hardware resources and energy, SoCs must nevertheless meet their application specifications.

With technology downscaling, efficient SoCs can be produced. However, the miniaturization brings on difficulties in manufacturing (increased number of *defects or imperfections*), variations, internal and external noise, such as: *crosstalks*, *electromagnetic interferences*, *impact of radiations* generating soft errors, and reliability problems such as *permanent defects* or *aging* [LSZ+09] [FRB08] [SK07] [SABR05]. All these may occur independently or they are correlated, some of them contributing to the apparition of others. Therefore, their (cumulated) effects are sometimes difficult to predict and prevent. Besides, the manifested symptoms of these phenomena make them quite challenging to diagnose. Undetected or untreated effects may result in logic or timing errors and ultimately they may lead to failures of modules or of the entire SoC.

Dependability is the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [IFI]. Even a perfectly designed system can fail if not all manufacturing defects, environmental factors and interactions during run-time are taken into account. Therefore, making the system dependable must be considered from the very beginning.

2.2 Networks-on-Chip

Global on-chip interconnects became a serious bottleneck for meeting the performance and power consumption requirements of complex, multi-processor systems-on-chip. The problem of global interconnects (mainly due to increasing delays and high power dissipation) is recognized as a challenging design constraint.

Traditional communication systems are dedicated *point-to-point* connections (P2P) and *busses*. Custom communication architectures (such as segmented buses with bridges) are also possible options for obtaining the maximum performance for a specific application. These and their hybrid versions work fine for relatively small systems, but present several problems for larger ones (several tens of nodes), such as: high wire density and length (P2P communication), complexity and high time to market for specific communi-

cation design (custom communication), and high delay of application messages (bus-based design).

The use of packet switched on chip interconnection networks, commonly referred to as *networks-on-chip* (NoCs), has emerged as the new on-chip communication infrastructure.

2.2.1 NoC Advantages and Drawbacks

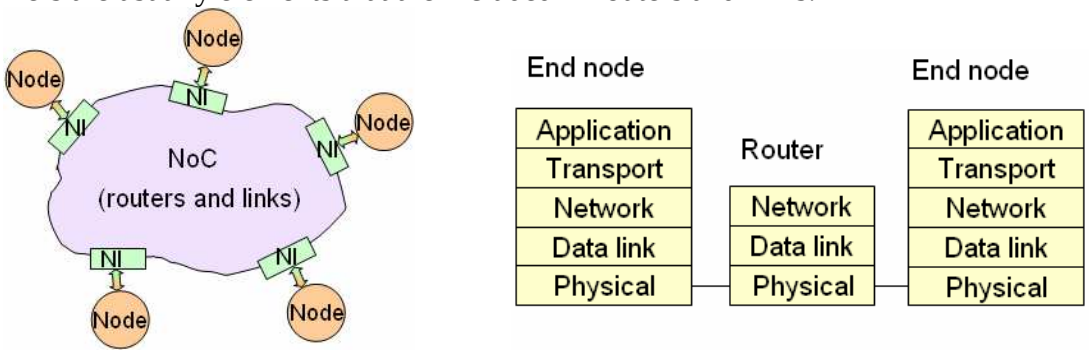
Face to the increasing complexity of SoCs, wire limitations (area, performance and power) and design and time-to-market constraints, NoCs brings in several *advantages*.

The NoC architecture uses *shorter links, efficiently shared*. A high level of *parallelization* is achieved, because all links in the NoC can operate simultaneously on different data packets. *Scalability*, *shorter design time*, *IP reuse* and *reduced time-to-market* are advantages that originate in the NoC *regular structure* and *modularity*; components (buffers, arbiters, routers, and links) and the communication protocol stack are easily *reusable*. NoC offers *predictable speed, electrical and physical properties*, based on its regularity. Another advantage is a *higher bandwidth*.

However, even if the NoC architecture is a very hot research topic at the moment, it is not yet widely adopted by the industry, mainly because of the following drawbacks: power, latency and the lack of tools and benchmarks. The power consumption is increased compared to the current bus-based approach, because of the complex *network interface* (*NI*) and switching/routing blocks, which are power greedy. The latency is higher because of the additional delay to packetize / de-packetize data at network interfaces and the delays at the numerous switching stages encountered by packets.

2.2.2 NoC-based System Elements and Communication Protocol Stack

In a NoC-base system, *nodes* communicate through the NoC and are connected to it using *network interfaces* (*NIs*), as depicted in Fig. 2-1.a. The NoC itself is an interconnection of *routers* and *links*; others of the above mentioned modules such as arbiters and buffers are usually elements that are included in routers and links.



a. NoC-based system b. NoC communication protocol stack

Fig. 2-1 NoC-based system (a) and NoC communication protocol stack (b)

Each of the NoC-based system elements is detailed subsequently.

- *Node*. A node represents an element of the SoC that communicates with other elements. It can be a cluster of nodes when the NoC is used as a global intercon-

nect or a simple element (general-purpose processor, DSP, memory block, peripheral etc.).

- *Network interface*, *NI*. It ensures the interface between the node and the NoC. The main role of the NI is to packetize/depacketize the messages, in addition to making the translation between different communication protocols. The basic unit of a packet is the *flow-control unit* (*flit*). A packet is formed of several flits: *header* (contains information about packet destination, length etc.), *payload* (data) and, possibly, a *tail* (end of packet).
- *Router*. A router has input and output ports connecting it to other components through links. Through the router, any input (or at least, a set of them) can be connected with any output (or, at, least, with a set of them). The main role of the router is to route incoming packets toward their respective destination in the topology. Additionally, the router has to perform arbitration among different port requests. Therefore, the router is implemented by means of buffers to store the incoming/outgoing data, switches and arbitration logic. More complex routers could also include a network processor.
- *Link*. A link connects two routers, or a router and a *NI*, or a router and a node directly. The implementation of a link can simply consists of data and control wires or it can be more complex, containing also buffers and additional logic [CGL+08].

The role of each layer in the communication protocol stack consists in offering services to the higher layer, by using the services of lower layers, while hiding its actual implementation to the higher layers, therefore allowing IP reuse. In the ISO-OSI standard [Zim88], seven layers exist. However, in most cases, it is not necessary to implement protocols at all of the OSI stack layers to provide high-level functionality. Usually, five layers are used for NoCs (Fig. 2-1.b):

- *Physical layer* refers to physical metal wires and devices that propagate and transform information. The physical layer protocols define signal voltages, timing, bus width, and pulse shape.
- *Data link layer* ensures a reliable transfer between adjacent nodes in a network, regardless of any unreliability in the physical layer (*logical link control* sub-layer, LLC) and deals with medium access (*media access control* sub-layer, MAC). The data link layer ensures that a connection is set up, divides output data into frames and handles the acknowledgements from the receiver when data arrived successfully. At the receiver side, the layer ensures that incoming data has been received successfully by analyzing bit patterns.
- *Network layer* is related to the topology and routing scheme over multiple hops. It provides a topology-independent view of the end-to-end communication to the upper level protocol layers. When data arrives at this layer, if it reached its final destination, it is formatted into packets and delivered to the transport layer. Otherwise, the data is pushed back to the lower layers to continue its route to the next hop. Network layer maintains logical addresses of the nodes in the network.
- *Transport layer* manages the end-to-end services and the packet segmentation/reassembly.
- *Application layer* is the interface between the applications running on the processing cores or the hardware IP cores and the communication architecture. Its role is to provide an abstract communication channel between IP cores, offering high-level communication services.

The link implements physical and data link layers and the router implements physical, data link and network layers. The *NI* hides network-dependent aspects from the transport layer.

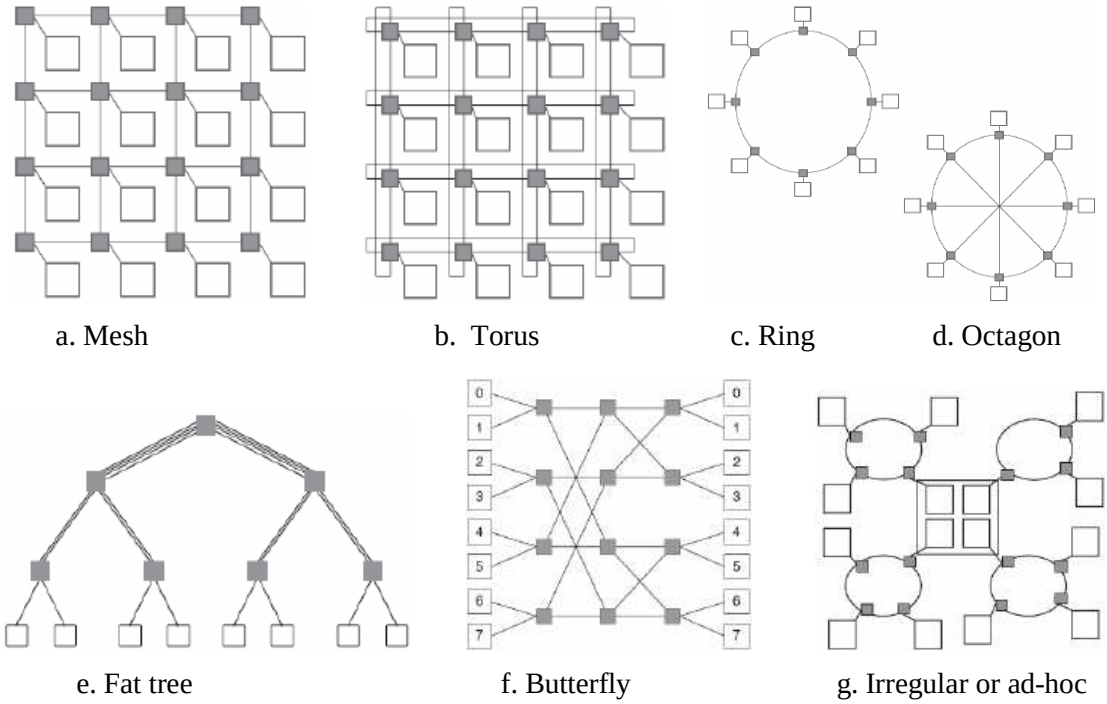
Regarding the flow control techniques, the main approaches are: *circuit-switching, store-and-forward* and *wormhole*, the latter being the most used in NoCs.

- In *circuit-switching*, a path is reserved between the source and destination nodes, during the whole transmission duration.
- In *store-and-forward* approach, the entire packet is stored in the router buffers before is forwarded to the next router or the destination core.
- In *wormhole* switching strategy, the flow is at flit level; once received at a router, the flit can be forwarded to the next node.

Each approach has its advantages and drawbacks. For example, entire packets can be checked at each hop in the store-and-forward approach, but large buffers are required at each router to store the packets.

2.2.3 NoC Topology

The *topology* specifies the way routers and links are interconnected. Some of the most used topologies are depicted in Fig. 2-2.



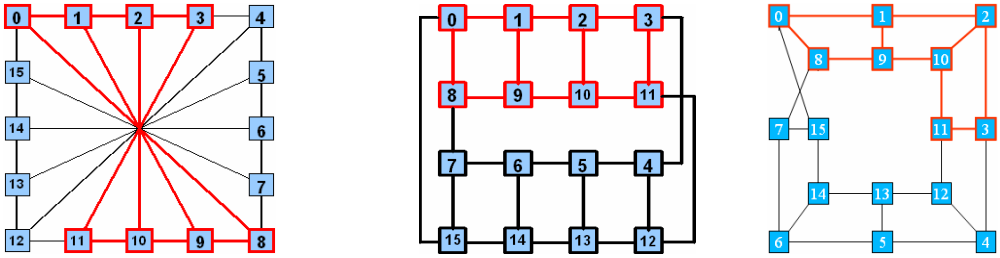
a. Mesh b. Torus c. Ring d. Octagon

e. Fat tree f. Butterfly g. Irregular or ad-hoc

Fig. 2-2 Different topologies

The choice of a topology depends on the advantages and drawbacks [TM09] [HB05]. Usually, regular topologies (Fig. 2-2.a-f) are preferred over irregular ones (g), because of their scalability and reusable pattern. Otherwise, irregular or mixed topologies can more conveniently be adapted to specific needs of the application. Because of certain constraints or optimizations (such as those related to place and route), a regular topology is not directly implemented physically as it is. A typical example is the ST Spidergon NoC [CGL+08] (Fig. 2-3.a), which looks physically as in Fig. 2-3.b. The number of long

links (across in Fig. 2-3.a) and the number and degree of crossing wires is limited, leading to a low-cost silicon implementation.



a. Regular topology b. Physical mapping choices

Fig. 2-3 ST Spidergon NoC topology

Depending on the paths the data follow through the NoC, a logic topology can be depicted, which can sometimes differ from the physical topology. For example, an octagon network could be configured to operate as a ring.

Different *routing policies* can be adopted, considering several criteria: the routing can be static or dynamic/adaptive, predetermined at source (router or NI) or distributed, implemented with logic or memories or software or a combination of these.

There are also other criteria to classify topologies. In NoCs, routers are connected to other routers through NoC links. Some of them also ensure IP connection to the NoC. In *direct* networks, each router is connected to one or several IPs, beside other routers. In *indirect* networks, routers can be connected only to other routers. Regarding the link types, topologies can be *undirected*, *directed* or *mixed*. In undirected graph topologies, all links are bidirectional, while in directed graphs they are unidirectional. The mixed-graph topologies are the most general case, where both types of links coexist.

Topology, routing, arbitration and switching policies, buffer dimensions, link width and critical paths of all components are some of factors that impact the NoC cost and the *quality of service*.

2.2.4 Quality of Service (QoS) in NoCs

Quality of service (*QoS*) of NoCs is measured in *latency* and *throughput* [BCGK04].

The *latency* is a measure of the delay that data (bits, flits, packets, or messages) experiences from its source node to its destination (sometimes the round trip is considered).

The *bandwidth* is the amount of data that can be physically and *reliably* transferred through a channel in a given amount of time. The basic unit of measurement for bandwidth is *bits per second* (bps), but it can also be expressed in kilobits, megabits and gigabits per second.

The *throughput* represents the amount of data that is actually *reliably* transferred through the NoC in a given time slot. The throughput is upper limited by the available bandwidth. The throughput can be affected by channel noise, errors, network topology and existing traffic in the network (data collisions, congestion etc.). It can be measured in bits per second (bps) or in data packets per second or per time slot.

As NoC is likely to become an attractive alternative for implementing SoCs for many application areas, some applications (e.g. real time and multi-media, such as audio/video streams) need strict deterministic bounds on delays and throughputs. In other applications, communication traffic is not bound by delay or throughput restrictions.

These two types of communication traffic are referred to as *guaranteed throughput* (*GT*) and *best effort* (*BE*) traffic.

A system containing concurrent applications may have concurrent GT and BE traffic in the network. In such mixed systems, the QoS of BE traffic tends to become low. Beside traffic requirements, some applications/tasks can be *critical* (e.g. automotive and security) and they require a higher degree of *reliability*. In some GT applications (such as multi-media), the reliability can be traded-off for the throughput. There are also cases where both requirements are important (GT and high reliability / data integrity).

Considering the different QoS requirements for different applications or tasks of the same application, the following question emerges:

Question: *How to achieve acceptable reliability levels with reduced impact on the QoS requirements by offering adaptability to the application specifics (data integrity, guaranteed throughput, best-effort)?*

This question is general; therefore it must be taken into account at each fault-tolerant mechanism that is implemented in the system.

2.3 System Dependability

This section details the dependability concept, considering several aspects: the *attributes* or *dimensions* (which are the ways to assess the dependability of a system), the *threats* that can affect the dependability of a system and the *means* – how to increase the dependability of a system.

2.3.1 Dependability Attributes

The original definition of *dependability* [ALRL04] for a computing system gathers the following non-functional attributes:

- *Availability*: readiness for correct service,
- *Reliability*: continuity of correct service,
- *Maintainability*: undergo modifications and repair.

Other attributes of dependability are [ALRV00]:

- *Safety*: absence of catastrophic consequences on the user and the environment,
- *Security*. Security is a composite of:
    - *Confidentiality*: absence of unauthorized disclosure of information,
    - *Integrity*: absence of improper system alteration,
    - Availability.

2.3.2 Fault-Error-Failure Chain

Dependability threats are *faults*, *errors* and *failures*. Definitions of these terms are given subsequently, and the relation among them is detailed.

A *fault* is a model of a defect in a system.

An *error* is a discrepancy between the expected behavior of a system and its actual behavior inside the system boundary. It can be seen as erroneous data or an invalid state.

A *failure* is an instance in time when a system displays behavior that is contrary to its specification. It is an observable deviation from the specified behavior at the system boundary.

The relation between dependability threats is known as the *fault-error-failure chain* [ALRV00]. The basic mechanism can be resumed like this: a fault may cause and error, which, in turn, may lead to a failure.

For example, a manufacturing *defect* in a logic gate can be modeled as a stuck-at-1 *fault* at the gate output, i.e., the output of the gate is always "1". However, the fault remains *latent* as long as the defective part of the component is not used; in our example, the fault is latent for all the inputs which produce correct gate output "1". The fault is *activated* when a deviation from the nominal state of the system occurs (deviation from the correct value of the output, in this example). An *error* consists in such a deviation – in our example, when the correct output of the gate would have been "0" but it is "1", because of the stuck-at-1 fault. An error can affect the behavior of other components in the system whose operation depend on the results of the defective component, and thus *contaminating* the system. If the contamination leads to a deviation of the external behavior of the system, unacceptable for the application, then a *failure* occurred. An occasional delay of an audio entertainment application can not be considered a failure, but a similar delay in the control system of a chemical installation can be catastrophic. The dependency fault-error-failure is depicted in Fig. 2-4.

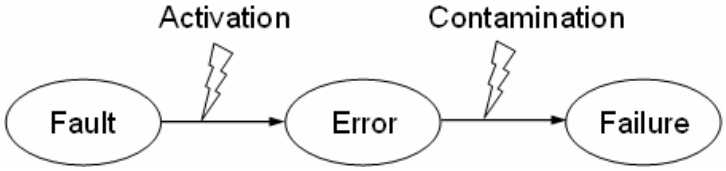


Fig. 2-4 Fault-error-failure mechanism

Since the output data from one service may be fed into another, a failure in one service may propagate into another service as a fault so a *chain* can be formed of the form: fault –> error –> failure –> error etc.

The fault-error-failure chain can be considered at different hierarchical levels (complete system, equipment, component, block, logic gate, transistor etc.) and apply both to hardware and software.

Two typical measurements for the sensitivity to failures are: *failure rate*, indicating the number of failures expected in the circuit in an interval of time and the *mean time between failures* (*MTBF*), indicating the average time between two successive failures. Sometimes MTBF is replaced by its variant, *mean time to failure* (*MTTF*), habitually used for components that can be replaced after the first failure.

2.3.2.1 Fault Models in Hardware

Faults are diversified considering their cause (specification, design and implementation mistakes, component defects, variability, internal and external noise), nature, place and rate of occurrence. Systematic approaches treat faults based on different models.

Basic *fault models* used at logic level include:

- *Stuck-at fault model*. A signal, or gate output, is stuck at a 0 or 1 value, independently of the inputs of the circuit.
- *Bridging fault model*. Two signals are connected together when they should not be. Bridging faults are normally restricted to signals that are physically adjacent in the design. Depending on the technology, either AND or OR logic gates can be used to obtain the resulting value of the wires.

- *Open fault model*. A wire is assumed broken, and one or more inputs of a block are disconnected from the output that should drive them.
- *Delay fault model*. The signal eventually assumes the correct value, but more slowly (or rarely, more quickly) than normal.
  Delay faults may produce variations of the critical path, therefore altering the maximum frequency specification. Possible causes of delay faults are process variation, crosstalks etc. [MRR97].

These types of models are used in all digital systems and they usually cover most of the defects and variations coming from HW.

Considering the fault nature, they can be *permanent*, or *temporary*. Temporary faults are further classified in *transient* and *intermittent*.

- *Permanent fault*. In general, a permanent fault always occurs when a particular set of conditions exists. In hardware, permanent faults model physical defects and they affect the circuit behavior always in the same way.
  Permanent faults in SoCs usually include the faults due to manufacturing processes defects. However, during normal operation, a number of wear-out mechanisms can occur in the long term, revealed initially as intermittent faults until they finally provoke a permanent fault [dARGG06].
- *Transient fault*. A transient fault perturbs the normal operation of an element in the system for a limited period of time; then, the affected element operation is no longer influenced. Usually they do not damage physically the hardware. However, the effects of a transient fault may last even after its duration ends.
  Transient faults in SoCs are mainly represented by the so-called *soft errors*. These are caused interferences of any kind, but also by cosmic rays and radiation impacts on silicon. Soft errors generated by radiation impact used to be reserved for harsh environments, such as space or nuclear power plants, with very high concentration of radiations or particles. However, with the advanced technology downscaling and the voltage and capability reduction, the circuit sensitivity to soft errors is significant even in normal environments (i.e. sea level). As a concrete example, Sun Microsystems use protection techniques against transient faults in their high availability servers, designed to be used at ground level [PC05].
- *Intermittent fault*. It is defined as a malfunction of a device or system that occurs periodically, either at regular or irregular intervals. Outside of these periods, the device or system functions normally. Generally, the cause of an intermittent fault is several contributing factors occurring simultaneously.
  Intermittent faults in SoCs are also likely to produce [Con08]. A possible cause is electromigration, whose first symptoms are intermittent glitches. Intermittent faults in interconnects typically cause burst errors.

Single or *multiple* faults can occur in the system. They can produce in one or several items in the circuit: transmission lines and combinational logic wire values can be delayed or inverted and register values and memory bits can be reversed.

2.3.2.2 Faults, Errors and Failures of Software

The faults in software originate in specification, analysis and implementation mistakes, as well as in external factors. Analogies with hardware domain can be done to model faults, errors and failures in software. The program code is analogous to the physical level; the values of the program state, to the computational level; and the software system running the program, to the system level. Therefore, a *bug* in a program is

considered as a *fault*, a possible incorrect value caused by a bug is an *error* and a possible crash of the application/OS is a *failure*.

Hang (the program becomes unresponsive to inputs) and *crash* (the program stops performing its expected function and responding to other parts of the system) are general software failures [MFVP08]. A few failure scenarios and causes are given below. Access to wrong parts of the memory such as *buffer overflow* or *dangling pointers* usually lead to software failure. Failures are also likely to occur when specific relative timing events of different components lead to unexpected behavior. Such *race conditions* are often hard to detect by testing only. Dependencies between processes can lead to *deadlock* (two or more processes wait for the other to finish) or *live-lock* (the states of processes constantly change, but none of the processes is progressing). *Resource starvation* is another example of software failure cause, which occurs when a process does not get the resource it needs, therefore it can never finish (e.g. the resource is constantly given to higher priority processes).

2.3.2.3 Hardware-Software Faults and Failures

There are several sources of faults that can impact the reliability of a NoC-based system. Some faults originate in hardware and others in the software running on cores [ST07]. However, faults in hardware induce errors or failures of software and vice-versa, as depicted in Fig. 2-5.

Consider a simple error occurring in hardware: a reversed bit. The place of occurrence can be anywhere in a NoC-based system, not necessarily in the NoC elements; errors can occur also in nodes (in processors, memories, internal communication structure of a node etc). The error can be produced by any type of fault. Such an error can manifest and affect the normal operation of the component, possibly leading to its failure.

The error can also be transported to other components of the system, by contamination. For example, if the error affects a register that is used during a branch condition execution, the error will change the program execution flow and can also propagate to the modules connected to this register. In a program stack, undetermined behavior of the program can result. Thus, a hardware error can directly produce errors and failures of software, hardware or both.

Software errors can lead to the failure of the entire system. It is also possible that erroneous software commands lead to physical destructions in hardware.

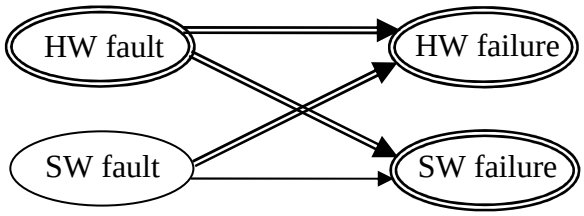


Fig. 2-5 Faults and failures in hardware and software

Hardware fault impact on NoC-based systems is considered in this thesis. Temporary faults that lead to data errors while transmitted through NoC links, as well as temporary faults that lead to software failure, are taken into account. Failures of NoC-based system components, independently of their causes (hardware or software), are also considered – i.e. node, router and link failure. Pure software faults are out of the scope of this thesis.

2.3.2.4 Faults and Failures in Communication Protocol Stack

Faults originating in lower levels of the communication protocol stack can lead to errors and failures in upper levels. Logic errors in the data link layer can affect the routing layer, for example: a reversed bit alters the data that is transmitted. If the packet address is altered, the packet will be misrouted or lost (when the altered address does not exist). Transport and application layers can also be affected by logic errors of data in packets. If packet loss, misrouted or altered cases are not anticipated at higher levels, failures can be experienced.

2.3.3 Failure Modes

Failure occurrences are usually avoided by dependability means. However, 100% failure-free systems can not be guaranteed. Instead, the failure mode must be controlled, especially for critical systems.

A *life-critical* or *safety-critical* system is a system whose failure may result in death or serious injury to people, loss or severe damage to equipment or environmental harm.

Fail-active systems seem to operate normally, but the operation is actually incorrect. A fail-active condition is a system malfunction rather than a loss of function. Fail-active mode can not occur in life-critical systems.

Several failure modes for life-critical systems are presented hereafter.

Fail-safe systems either behave correctly or stop after an internal failure is detected. The failure *allows* but does not *cause* or *invite* a certain improper behavior. Many medical systems fall into this category. Railway signaling is designed to be fail-safe, such as any failure is translated into a stop signal. Elevator cabins have a safety mechanism which uses contact with the guide rail to decelerate the car so they do not drop if the cable breaks. During early Apollo program missions to the Moon, the spacecraft was programmed on a free return trajectory; if the engines failed at lunar orbit insertion, the craft would safely coast back to Earth [Dum01].

Fail-secure systems maintain maximum security when they can not operate. Failures do not allow a certain improper behavior, although some proper behaviors are impeded. An example of fail-secure system in banking domain is an ATM system which stops delivering money after a failed operation.

Fail-stop systems always halt on any internal failure, before the effects of the failure become visible. Fail-stop processor design is addressed in [SS83]. Usually, the halt is indicated by an error signal at the system outputs.

Fail-silent systems behave similarly to fail-stop systems, but no error or warning about the failure is produced at outputs.

Fail-operational systems continue to operate when they fail. As example in avionics, a fail-operational automatic landing system is designed so that in the event of a failure, the approach, flare and landing can be completed by the remaining part of the automatic system.

Fail-passive systems continue to operate when system failure occurs. An example includes an aircraft autopilot. In the event of a failure, the aircraft would remain in a controllable state and allow the pilot to take over, complete the journey and perform safe landing.

These failure modes can be used at the system level but also at sub-systems, to limit the failure impact on the rest of the system.

2.3.4 Dependability Means

Dependability means are intended to reduce the global number of failures of a system. Considering the mechanism of fault-error-failure chain, it is possible to propose means to break these chains and thereby improve the system dependability. Four means are identified:

- *Fault prevention*: consists in preventing faults being incorporated into the system. Specification, design and implementation phases are mainly concerned. Faults are prevented by employing best known and already validated design methods, module reuse, structured programming etc.
- *Fault forecasting*: predict faults likely to occur (using mathematical models), so that they can be removed and their effects circumvented.
- *Fault removal*: Faults can be removed either during development or during run-time (e.g. by HW/SW testing).
- *Fault tolerance*: Adding mechanisms in the system allowing delivering the required service even in the presence of faults (sometimes at a degraded performance.

Dependability means must be considered cooperatively. For each type of fault, there are several methods to implement these means; the choice depends on application and dependability level associated with it, performance penalties and overall costs of these methods.

Techniques to improve the functional yield, parametric yield, and reliability of ICs are nowadays considered early in the design stages. They are grouped under the name of *design for manufacturability* (*DFM*). Examples include checking and correcting electromigration issues at layout-level, shielding to deal with electromagnetic interferences and crosstalks. Some manufacturing defects can be detected at the post-fabrication test phase. *Latent* defects (e.g. a partial void formation) are not likely to be detected at this phase; they will manifest later, during the lifetime of the product. Faults due to aging or variability are likely to happen only under certain scenarios that have small probability to be reproduced in the test phase, because of the huge amount of possible combinations of inputs and states of the system (which make exhaustive tests unacceptable long). Aging effects usually occur after a certain time of utilization of the circuit, so they will produce later. Soft errors are likely to occur anytime during the normal operation conditions of the system and affect randomly any part of the system. They can be forecasted and treated during run-time.

Similar means are used to deal with software faults: *prevention*, *removal*, *fault-tolerance*, and *input sequence workarounds* [Wil00]. *Prevention* is done by using formalized and structured design methodologies and technologies aimed at preventing the introduction of faults into the design. The proper use of software engineering during the development processes is a way of realizing fault *prevention* and *removal*. Lexical, syntactic and semantic faults in software code are detected and removed during compilation (e.g. misspelled, undefined or not initialized variables) and debug. Other faults are detected by software reviews, analysis and tests and then *removed*. Bugs are faults that manifest only on specific inputs or conditions, therefore they are not easy to detect and correct.

2.3.5 Fault-Tolerant Solutions

2.3.5.1 Fault-Tolerant Solutions in Hardware

Fault-tolerant techniques are generally classified in *static* and *dynamic* (Fig. 2-6).

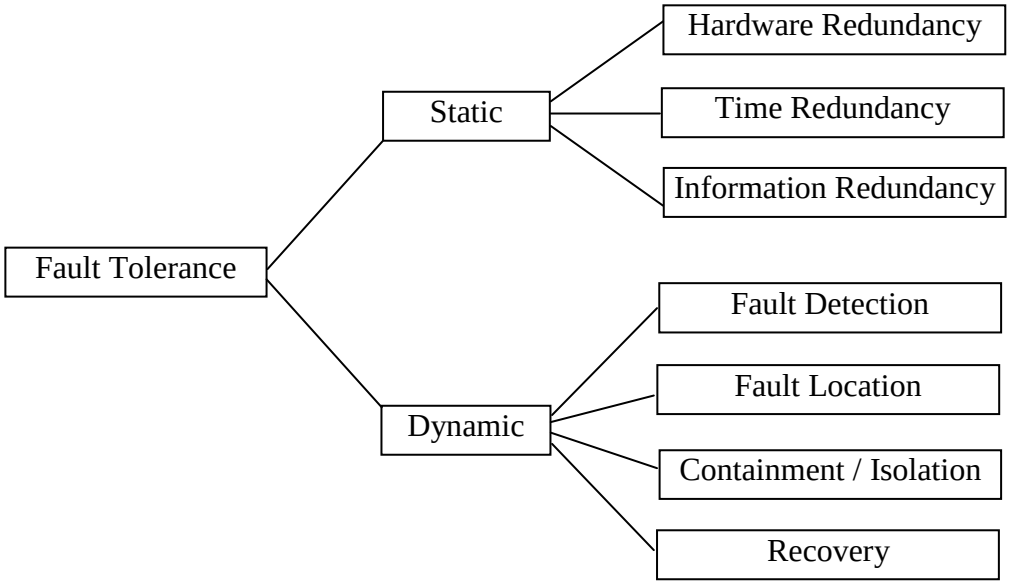


Fig. 2-6 Fault-tolerant solutions

Static fault-tolerance is built into the system structure and it effectively masks the fault effects. It can be realized by hardware, informational or temporal redundancy. Classifications by redundancy types are often present in the literature. However, in most cases, redundancy types are mixed. For example, error correction/detection codes are seen as information redundancy, but they also imply hardware redundancy to store the additional data and to perform the coding/encoding.

Triple modular redundancy (*TMR*), and its generalized version *NMR*, consist in using three (odd N, respectively) identical modules instead of one. Their results are submitted to a voter. The majority decides the final result. This method is efficient when the probability that the majority of modules are affected by identical errors in the same time is negligible. NMR is an expensive method and its critical part resides in voter dependability; if this fails, the whole system may fail. NMR can be applied at any hierarchical level of the architecture. It can be also applied in the software – *task replication* is an example.

A version of TMR translated to the *time* domain consists in repeating the operation several times by using the same module and performing voting among the results. This method deals with transient and intermittent faults. It is applied when speed is not critical and hardware overhead has to be kept to the minimum. Variation of this technique can use different coding of data for each retry to mask permanent faults in addition to temporary faults. TMR and its extended versions implement mainly the fault masking.

Dynamic fault-tolerance is achieved by active operations, as opposite to the passive nature of the static fault-tolerance. It consists in several phases: fault detection, location, containment, isolation and recovery [GAP+07].

- *Fault detection* is essential in dynamic fault-tolerance. Undetected faults can affect a system or component outputs that are falsely considered as fault-free. The

basic implementation consists in duplication (hardware or temporal) with comparison. More elaborated implementations use *error detection codes* (*EDC*): *parity*, *dual rail*, *Hamming*, *BCH*, *M-out-of-N*, *Berger*, *arithmetic* codes. The choice of the proper code is a critical task and must be done considering the targeted reliability (type, multiplicity and pattern of errors detected) and the area, power and timing overhead implied. Fault detection can also be implemented as *periodic tests* or using *watchdog timers*.

- *Containment* and *isolation* refer to avoiding the propagation of the fault consequences (the error) and limiting thus its impact in a *fault containment region* (*FCR*). Containment and isolation of errors are realized for instance at operating system level using task-structured applications, where each task has access only to a set of resources and can not corrupt the resources of other tasks.
- *Recovery* is the ultimate goal of fault-tolerant schemes and provides means to recover from failures. Recovery solutions include: operation/transmission *retry*, *error correction codes* (*ECC*), and *spare modules* with static or dynamic *reconfiguration*, *masking*, *repairing* and *reset*.

Like the fault-error-failure chain, fault-tolerant solutions can be applied at different hierarchical/abstraction levels, all in the same level or distributed in across them. In *hardware*, fault-tolerant solutions are implemented at the following levels: structures of the transistors and manufacturing process, technological cells, logic gates, computing primitives, micro-architecture, and global architecture.

2.3.5.2 Fault-Tolerant Solutions in Software

Considering the complexity of software, software validation and verification is very difficult. However, software failures can be crucial for the entire system reliability; therefore fault-tolerant mechanisms implemented in *software* are mandatory. These mechanisms complement the hardware ones, especially when not enough protection is provided at hardware level (usually because of high costs). Protection is implemented at different levels of the software stack: *operating system* (*OS*), middleware, and/or application levels.

Single version software fault-tolerance techniques include *system structuring and closure*, *atomic actions*, inline *fault detection* and *exception handling*. A few other fault-tolerant techniques are mentioned hereafter, showing their similarities with the fault-tolerant techniques used in hardware. *Recovery blocks* in software are similar with reconfiguration using spare components in case of error detection. *N-version programming* is similar with NMR with different versions. It implies developing different versions of the program, using different algorithms, programming techniques and languages and different programmer teams, in order to avoid the presence of the same *bugs* affecting all program copies in the system. *N self-checking programming* uses either acceptance tests [Par96] or duplication with comparison for each version of the software. Other fault-tolerant mechanisms can be obtained by combining the above-mentioned ones [Wil00].

Methods similar with TMR or duplication with comparison, i.e. *task replication* (simple or combined with n-version programming), are quite expensive. The main impediment of these techniques is the high overhead, not only temporal (when the versions are successively executed on the same processor) or hardware (when several processors are used) but also because of the important size of software that has to be stored.

Using *input sequence workarounds* is the last line of defense against faults. Examples include not entering a particular input sequence to which the system has demonstrated susceptibility or entering series of inputs trying to return the system to an acceptable state. The ultimate workaround is the restart. However, a system exhibiting such behavior is not dependable.

In the last years, an increased attention has been given to runtime fault-tolerant techniques, because of the increased probability of transient faults, which can corrupt the consistency of the data and instructions stored in memory or handled by the program.

Checkpoint and Rollback Recovery (C&R) is a convenient fault-tolerant solution to deal with temporary faults, as well as with IP failures. In the latter case, upon failure detection, the task is migrated to another IP. C&R can be implemented either at the OS or at the application level. Its principle is to periodically store consistent states and rollback to a previous saved state in case of fault detection.

Different C&R techniques were studied for distributed systems [KT87] [GR06]. The main approaches are *uncoordinated* and *coordinated checkpointing*, presented in details in chapter 4. Uncoordinated checkpointing incurs lower overhead during the failure-free execution, thus allowing higher local checkpointing frequency, while the recovery is more complex and prone to the domino effect. On the other hand, coordinated checkpointing is preferable in practice, considering its simplicity in synchronizing a global, consistent fault-free state, as well as its recovery protocol simplicity. However, with the increasing number of PEs, C&R techniques, especially coordinated checkpointing, suffer from poor scalability [EP04].

C&R techniques become relevant for multi-processor systems-on-chip, where the number of PEs is projected to considerably increase in the near future (e.g. hundreds or even thousands) [ITR09a]. Considering also the increased number of faults that appear with the significant miniaturization of the integrated systems, improving the scalability of the recovery by checkpointing becomes decisive. In this context, the following question arises:

Question: *How the checkpoint and rollback mechanism impacts on the NoC performance can be evaluated and how its scalability can be improved?*

Answers to this question are given in Chapter 4.

2.4 3D Integration and 3D NoCs

"2D scaling will slow considerably. When this happens, we need to exploit the 3D to stay on the performance growth curve." (Mike Ignatowski – IBM) [LL08]

Even benefiting of the continuous technology down-scaling, as they become more and more complex, the systems-on-chip keep on increasing their area, power consumption and communication latency. Moreover, downscaling seems to be limited by the wire delays. With larger chips, several additional problems must be worked out, such as clock distribution and signal integrity. Also, the latencies between marginal nodes increase, even when a NoC is used for the communication, since many hops have to be traversed.

In this context, the 3D integration paradigm seems to be a promising solution to cope with very large chip problems [BSKS01] [DCR03]. There are several candidate variants on 3D integration technology. Building *monolithic* 3D chips consists in building electronic components and their connections in layers in a single semiconductor wafer, which is then diced into 3D ICs. Applications of this method are currently limited because creating normal transistors requires enough heat to destroy any existing wiring.

Therefore, the actual manufacturing technique consists in building the electronic components in several wafers/dies that are then *stacked* [Li08], as depicted in Fig. 2-7 [Led07].

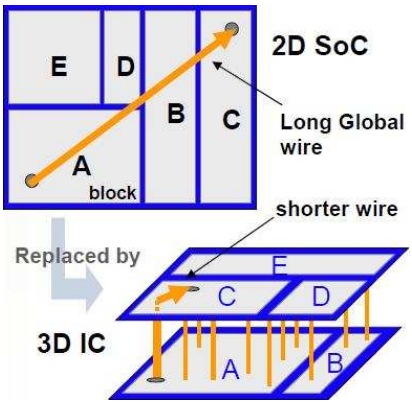


Fig. 2-7 3D vs. 2D chip (Source P. Leduc)

The 2D layers of the stack communicate through vertical links, called *die-to-die vias*, *3D vias* or *through-silicon-vias* (*TSVs*). Replacing long horizontal with short vertical interconnects addresses RC delay, crosstalk and power consumption. The number of TSVs that can be allowed across any two layers strongly depends on the underlying 3D fabrication technology [SMBM09].

Unlike a *system-in-package* (SiP), where the chips in package communicate through off-chip interconnects, a 3D IC will be a single chip; all components in the layers communicate through on-chip interconnects, whether they are vertical or horizontal.

2.4.1 3D Integration Advantages and Challenges

The main **advantages** offered by this technology are summarized in the following paragraphs.

Footprint – More functionality fits into a smaller space; the device density increases. This extends Moore's Law and enables a new generation of tiny but powerful devices.

Speed – The average wire length becomes much shorter. Because propagation delay is proportional to the square of the wire length, overall performance increases.

Power – Keeping a signal on-chip reduces its power consumption by ten to a hundred times [Dal06]. Shorter wires also reduce power consumption by producing less parasitic capacitance. Reducing the power budget leads to less heat generation, extended battery life, and lower cost of operation.

Bandwidth – The bandwidth is increased compared to SiP.

Heterogeneous integration – Circuit layers can be built with different processes, or even on different types of wafers. This means that components can be optimized to a much greater degree than if they were built together on a single wafer. Even more interesting, components with completely incompatible manufacturing could be combined in a single device [Led07] [DM09], enabling new functionalities.

Beside the direct advantages, the 3D integration opens several *new possibilities* for:

Design – The vertical dimension adds a higher order of connectivity and opens a world of new design possibilities.

Routing – More flexibility to route signals, power and clock.

Circuit security – The stacked structure hinders attempts to reverse engineer the circuitry. Sensitive circuits may also be divided among the layers in such a way as to obscure the function of each layer [Tez08].

Since 3D technology is quite new, it carries new **challenges**. There are several options which are currently explored by different research groups and fabricants, concerning the bonding/dicing order, the adjacent faces of layers, the TSV realization moment, different TSV technology and materials etc. The most important challenges are achieving acceptable levels of yield, heat dissipation, and design complexity management across the layers.

Yield – Each extra manufacturing step (layer thinning, TSV creation, bonding) adds a supplementary risk for *defects*: misalignment, dislocation, void formation, oxide film formation over copper interfaces, pad detaching, defects due to temperature, coupling and so on. In addition, the cumulated effects of these defects are very difficult to predict and prevent.

Heat – Thermal buildup within the stack must be prevented or dissipated. Different solutions have been proposed, including thermal TSVs [WL06].

Design complexity – Taking full advantage of 3D requires intricate and elegant multi-level designs. Chip designers will need new CAD tools to address the 3D paradigm [CS09].

TSV footprint – The footprint of a vertical link is huge with respect to the 2D counterparts, because of the very large TSV diameter and pitch (tens of μm [ITR07] [DDF+08]).

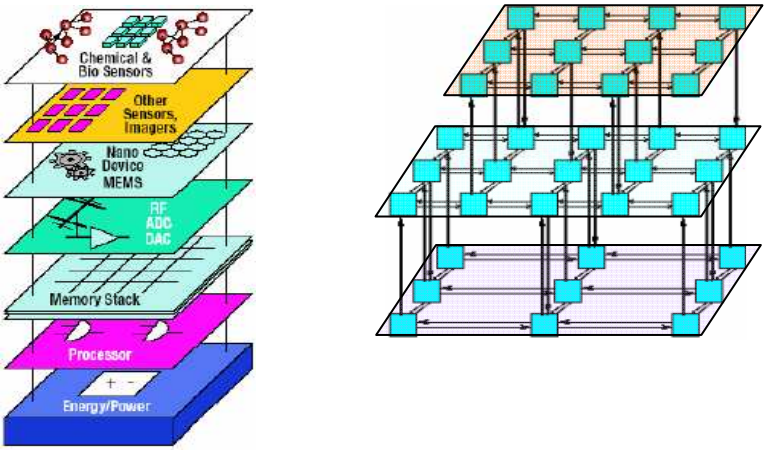
Beside these challenges, the time-to-market is an important constraint. While still searching for solutions to cope with these challenges, 3D chips with 4-5 layers and 3D memories with 8 layers [Sca07] have already been realized.

2.4.2 3D Network-on-Chip Topology and Routing

3D integration paradigm combined with NoC approach (3D NoC) seems to be the most promising solution to cope with very large chip problems [BSKS01] [DCR03]. The 3D NoC vertical links consist in bundles of TSVs integrated in the communication system by so-called *vertical* or *3D routers* [BSS09]. The corresponding nodes are called: *3D* or *vertical nodes*. 3D routers are manufactured in 2D layers and have extra-ports to/from adjacent layers. Nodes and their routers with no such ports are called: *2D nodes* and *2D routers*, respectively.

There are two main reasons for which not all nodes can be 3D nodes; these are the heterogeneity of layers and the restrictions regarding the TSV (size, number, and yield, mainly).

Heterogeneity is one of the biggest advantages of the 3D integration. Layers of the 3D stack can be built with different technologies for each layer (digital, analog, RF, MEMS, chemical and bio sensors etc.) – see Fig. 2-8.a. They communicate with each other, but they do not necessarily have the same topology. Even for the same type of topology (e.g. mesh) in two layers, the NoC link length can be different for the two layers, depending on the respective IP count, size and placement. As a consequence, not all nodes in a layer will be connected by vertical links. Besides, 3D routers occupy larger areas than 2D routers, because of their higher grade of connectivity (7x7 ports vs. 5x5, for 2D meshes) and the minimum pitch allowed by a specific TSV technology. Restrictions related to the TSV count, size and positions also impact the stack vertical connectivity degree [DM09]. Moreover, signal, power/ground and thermal TSVs all compete for area and routing resources.



a. Heterogeneous 3D integration b. 3D NoC topology

Fig. 2-8 Heterogeneous 3D integration and 3D NoC topology

In these conditions, a partially vertically connected NoC topology (Fig. 2-8.b) is an efficient solution to the utilization of a reduced the number of TSVs and meanwhile to adapt to the heterogeneity and the irregularities of 3D systems.

Beside transient and intermittent faults produced during runtime, permanent failures of some components of the NoC-based system (such as links, routers, IPs) can also occur, during or after fabrication [Fur06] [HJS04]. According to safety requirements, a single failed component should not cause total system failure. A failed element must be deactivated and the system must reconfigure [AN05] and be able to still deliver its services, even if at a degraded performance. Besides, reconfiguration capability should allow safely shutting down chip regions and readjusting traffic patterns upon a change in the software application running on the chip.

Considering the 3D NoC topology characteristics (incomplete vertical connection and possibly different topologies in each layer), as well as the possible failures of routers and links, the following question rises:

Question: *How to route messages in 3D NoCs with partial vertical connectivity and possibly different layer topologies, with minimal design effort, overhead and time-to-market, while ensuring an optimal routing inside each layer?*

Taking into account the poor yield of TSVs, methods to improve the link dependability must be considered before declaring them as failed. Spare wires and serialization are currently used to deal with permanent faults. This aspect is not addressed in this thesis.

Question: *How to deal with transient and intermittent faults in NoC links?*

Answers to these questions are discussed in chapters 5 and 6, respectively.

2.5 Conclusions

General aspects and problems related to the dependability of networks-on-chip are presented in this chapter. Considering the tight collaboration among system elements, we conclude that fault-tolerant mechanisms are necessary for all components and at all levels of the communication protocol stack, in order to limit the propagation and effects of unavoidable faults and errors, and to obviate the failure of the entire system. A fault-tolerance capability vs. performance / cost trade-off must be considered for each type of application, according to its specific QoS. In this context, adaptability and reconfiguration are key points. Following this analysis, several questions have emerged:

Question: *How to achieve acceptable reliability levels with reduced impact on the QoS requirements by offering adaptability to the application specifics (data integrity, guaranteed throughput, best-effort)?*

Question: *How the checkpoint and rollback mechanism impacts on the NoC performance can be evaluated and how its scalability can be improved?*

Question: *How to route messages in 3D NoCs with partial vertical connectivity and possibly different layer topologies, with minimal design effort, overhead and time-to-market, while ensuring an optimal routing inside each layer?*

Question: *How to deal with transient and intermittent faults in NoC links?*

Answers to the last three questions are discussed in the following chapters, taking into account answers to the first one.

Chapter 3. State of the Art of Fault-Tolerant Techniques in NoCs

---

This chapter presents the state of the art of fault-tolerant methods implemented in NoCs. The first part concerns techniques that achieve application recovery from errors or failures by means of checkpoint and rollback. Failures of NoC elements (routers and links) are addressed by fault-tolerant routing algorithms, overviewed in the second part of the chapter. Before declaring a NoC element as failed, fault-tolerant techniques are used to improve its reliability. Methods to tolerate transient faults during NoC transmissions are presented in the last part of the chapter.

3.1 Checkpoint and Rollback Recovery

The works presented in this section address checkpoint and rollback recovery in distributed systems and computer clusters. These techniques become relevant for MP-SoCs with high number of processing cores where failures are more and more probable to occur.

3.1.1 Principle, Concepts and Models

The rollback recovery based on checkpoint is used for recovery after application failures caused by temporary (transient or intermittent) faults, as well as for porting an application between homogeneous or heterogeneous nodes [SFJ03] [JKG09] [LMS+05]. Porting is useful to recover processes after the failure (crash) of the core they were running on.

The recovery is based on application intermediary states. Information necessary and sufficient to reconstruct a consistent state of the application is periodically saved during the error-free execution. This information (*checkpoint*) is to be used in case of failures, to resume the application. The state that the application is resumed from must be consistent in order to ensure correct resuming. An application state is *consistent* if it could be reached following a correct and error-free execution of the application from the start point. Factors that can lead to saving corrupted states are analyzed in [CC02]. Acceptance tests [Par96] are used before saving application states, to avoid saving corrupted states.

The checkpoints are saved on a memory whose content persists despite potential faults and called *stable storage*. It can be either global, i.e. localized in a single node and therefore shared by all nodes, or distributed at each node. Some implementations use only *permanent checkpoints* (directly saved on stable storage), while others make use of *tentative checkpoints* stored on the volatile memory and written in stable storage only after being validated; otherwise they are discarded. *Mutable* checkpoints have also been proposed to reduce the overhead in mobile computing [CS01]. Not all of the checkpoints saved on stable storage are certainly used for application recovery. Old or unnecessary checkpoints are removed using *garbage collection* mechanisms, to free the stable storage for future use.

A few of the most used assumptions in checkpointing and rollback implementations are summarized hereunder [EJZ92]. Processes communicate only by exchanging messages. The system is asynchronous: there exists no bound on the relative speeds of processes, no bound on message transmission delays, and no global time source. Hence, the order in which a process receives messages is nondeterministic.

The execution of a system of processes is represented by an irreflexive partial ordering of *send*, *receive* and *delivery* events ordered by potential causality [Lam78] [EJZ92]. Delivery events are local to a process and represent the delivery of a received message to the application. Channels are usually assumed to be FIFO-like (i.e. messages are received in the order they were sent). Furthermore, a sent message will eventually be received, if the receiver does not fail.

Fail-stop or *fail-silent* failure model is assumed for the processes. A fail-stop processor halts instead of performing erroneous transformations. Others can detect halted state and can obtain uncorrupted stable storage content. The volatile memory content is lost when a failure occurs [SS83] [Sch84].

3.1.2 Solutions based on Checkpointing and Message Logging

Different checkpoint techniques were studied for distributed systems [KT86] [GR09]. The main approaches for saving the intermediary state (checkpointing) are *coordinated* and *uncoordinated*. In the coordinated checkpointing approach [CL85], the application tasks synchronize in order to establish and save a consistent global recovery line. In case of failure, tasks resume execution from the most recent recovery line. In the uncoordinated checkpointing approach, different tasks take their checkpoints independently and, in case of failure, the most recent recovery line must be determined before resuming execution from it. This approach induces small overhead during the failure-free execution, but it is prone to the *domino effect* [Ran75] [KT86] in case of failure, when tasks force each other to rollback in order to reach a consistent recovery line. In the worst case, the application rolls back in time to the starting point.

Message logging is intended to limit the domino effect. It helps in forming a consistent recovery line by using individual checkpoints taken by tasks at different moments in time and which otherwise do not form a consistent state. Thus, a task checkpoint consists of the task state and, possibly, some logged messages necessary in case of recovery. The logging of messages is *optimistic*, *pessimistic* or *causal* [AM95]. In the optimistic approach, the messages can influence the system state before being logged, while in the pessimistic one, they are logged first. Each approach has its advantages and drawbacks. In the optimistic case, smaller latency is incurred during fault-free execution, but the recovery is ampler when compared to the pessimistic case. Causal log was designed to improve fault-free performance of pessimistic log. Messages are logged locally and causal dependencies are piggybacked to messages. The overhead of causal log is still non negligible, but slightly smaller than in the pessimistic case. The fault recovery overhead is limited, but greater than for the pessimistic log. Another option in message logging concerns the end node where the messages are saved: at sender or at receiver. *Sender-based* logging requires no stable storage, but incurs higher number of messages exchanged between sender and receiver. Another problem is that the previous state can not be recovered in some cases of concurrent failures, even if a pessimistic logging is performed. *Receiver-based* logging requires stable storage [AHM93].

A comparison between coordinated checkpointing and message logging for large clusters is addressed in [LBKC04]. The conclusion of this study is that coordinated checkpointing performs better when fault frequency is low; the application execution stops advancing for one fault every hour, for the application considered in this work. For higher fault rates, message logging becomes more effective. Algorithms and implementations derived from the basic protocols have been also proposed, for distributed and parallel systems.

Several efforts were directed in studying *communication-induced checkpointing* (CIC). CIC protocols were proposed as an improved approach combining coordinated and uncoordinated checkpointing protocols. In order to enhance the overall checkpointing performance, CIC tempts to reduce the number of tasks participating to the global checkpointing by dynamical analysis of traffic pattern. However, a more recent work [AER+99] shows that CIC protocols do not scale well with larger number of tasks. Hierarchical checkpointing protocols have also been proposed for large clusters and cluster federations using coordinated checkpointing or combining coordinated checkpointing and CIC [MMB04]. An excellent overview of rollback recovery in message-passing systems can be found in [EAWJ02].

3.1.3 Adapted and Optimized Solutions

Because of its simplicity in synchronizing a global consistent fault-free state, coordinated checkpointing is preferable in practice. Choosing appropriate checkpointing interval (i.e. frequency) is one of the factors that highly influence the application performance. More often checkpointing reduces system recovery time. On the other hand, frequent checkpointing increases latency during fault-free execution. To determine an optimal checkpointing interval, minimum impact on the system overall quality must be considered [CYR09] [HKC+01].

The main *scalability limitation* of the coordinated checkpointing is the duration of the global synchronization. With the increasing number of PEs, rollback recovery by coordinated checkpoint suffers from poor scalability [EP04]. One of the causes is the contention at the storage system, in the case of a common storage element, since all tasks take their checkpoints in the same period of time. Besides, with the increase of network size, checkpointing duration increases. This is because global synchronizations necessary to build global consistent state may induce significant communication overhead for large systems or they are delayed by existing traffic in the network. Since the protocol allows no overlapping of checkpointing phases, the possible interval of time between two consecutive checkpoints enlarges (being lower bounded by the checkpoint duration). In consequence, after a rollback, re-bringing the application to a failure-free state incurs greater latency. This situation becomes even more drastic when the synchronization latency cannot be reduced below the expected interval between two consecutive failures. In such cases, there is no sufficient time to take new global checkpoints; therefore rollbacks to already old checkpoints are performed in case of failures. When the probability of occurrence of such conditions is high, the application stops advancing the execution.

Different approaches were proposed to reduce the overhead of coordinated checkpointing algorithms, such as minimizing the number of synchronization messages and the number of checkpoints, making the checkpoint non-blocking or combinations of these [CS98]. Group-based coordinated checkpointing for reducing contention at the common stable storage has also been proposed [GHKP07]. These techniques become relevant for multi-processor systems-on-chip (MPSoC), where the number of IPs is projected to increase dramatically (in the order of hundreds/thousands) in the near future [ITR09c]. Considering also the increased number of faults that appear with the significant miniaturization of the integrated systems, improving the scalability of the recovery by checkpointing becomes decisive.

Several *blocking* and *non-blocking* coordinated checkpointing algorithms have been proposed [KT86] [BCH+08] [GR09]. In the blocking approach, tasks block their normal execution to perform the checkpoint, save it on stable storage, send an acknowledgement to the checkpoint initiator and wait for the commit. They resume execution only when they have received this commit. The commit is sent by the initiator when it has received all the acknowledgements from all the computing nodes. In the non-blocking approach, tasks overlap their normal execution with the synchronization phase. Generally, the non-blocking approach is preferred because of its lower latency. Thus, there are many works dedicated to proposing new or improved non-blocking protocols for distributed and parallel systems [GR09] [QS03]. A comparison between blocking and non-blocking protocols is presented in paper [BCH+08]. Both protocols considered in this work rely on the Chandy and Lamport algorithm [CL85]. In this algorithm, one or more processes can initiate a checkpoint wave. When a process starts a checkpoint, it records its local

state and sends a marker to all its neighbors. Upon the receipt of a marker, a process starts its checkpoint wave. Messages received after having started its checkpoint wave and before having received the marker of the sender are recorded in the receiver images as the channel's state. The first protocol considered in [BCH+08] is a direct implementation of this algorithm, while the second consists in synchronizing the different processes for emptying the communication layer. Thus, during the checkpointing, no messages are exchanged; so there is no need to store the channel state. *Blocking* and *non-blocking* of the process while the checkpoint is written on the stable storage are also options that must be considered for minimal penalty on performance [EJZ92]. *Incremental* checkpoint has also been proposed [FL06]. After a checkpoint is created, state changes are logged incrementally as records in memory. Small and constant time is required for checkpoint creation and state changes recording.

Topology characteristics have also been considered to reduce the checkpoint and rollback recovery overhead. Algorithms for checkpointing and rollback recovery in asynchronous unidirectional and bidirectional ring networks are proposed in [MM04]. In [GMRV06], an improved checkpointing algorithm for unidirectional ring networks is proposed. The algorithm does not take temporary checkpoints and has a reduced duration of checkpoint.

The checkpoint and rollback recovery can be implemented in different manners: *explicit* or *application-level* (managed by the programmer), *semi-automatic* (guided by the programmer) and *automatic* or *system-level* (transparent for the programmer/user) [BCH+08] [BPS06] [dCGKG04] [GYHP06]. System-level checkpointing has the advantage of transparency from the programmer/user point of view, but incurs higher overhead induced by its generality. Application-level checkpointing has the benefit of smaller overhead, at the price of implying the programmer in tailoring the fault-tolerant implementation for the application specifics.

3.2 Fault-Tolerant Routing

Many routing algorithms have been proposed for 2D NoCs, considering both general and NoC-specific requirements: adaptability, congestion avoidance, low latency, low overhead, low power consumption, fault-tolerance. Regular 2D topologies (mesh / torus, spidergon, and ring) are generally considered, for which routing algorithms are simpler and incur lower overhead. A few fault-tolerant routing algorithms have also been proposed for regular 3D mesh and torus topologies.

3.2.1 Routing Classifications and Optimizations

The routing function can be simply resumed as being the mechanisms that ensures that packets reach their destinations in a finite interval of time, by steering them through intermediary nodes in the network. Besides the main goal, of leading packets to their destination, there are two main requirements that a routing algorithm must meet, to ensure the temporal constraint: it must be *deadlock-free* and *livelock-free*. A deadlock occurs when packets can not advance because of direct or indirect dependencies among their paths. A livelock occurs when the packet is moved through the network in order to reach its destination, but it never reaches it.

When designing a routing algorithm, the efficiency-cost trade-offs must be taken into account. There are several factors to take into account: topology regularity, hardware-software partitioning of routers, application specifics, expected failure rate etc. The

routing is either *deterministic* (static or oblivious [Rac09]) or *adaptive* (dynamic). In the deterministic routing, the route between a source and a destination node in the network is always the same, while in the case of adaptive routing different routes may be proposed for the same source-destination pair of nodes, depending on the network instantaneous traffic. Other classifications are done considering *where* and *how* the routing decision is taken. Thus, the route to the destination is determined either at the source node or along the path (*distributed routing*). Also, the routing is either based on an algorithm or on routing tables.

Various choices about the routing approach (static/dynamic, source-based/distributed, algorithmic/routing tables) are discussed hereafter, regarding the NoC regularity degree.

Deterministic routing algorithms are very simple for regular topologies, so the cost of embedding them in each node is smaller when compared to a routing table. Thus, the route is determined on the fly, at each node, reducing thus the overhead of a predetermined route that must be carried together with the message (this can be long for large NoCs). On the other hand, the latency incurred by a complex routing algorithm execution at each node may be higher when compared to retrieving the next node from the route carried along with the message.

However, as the NoC irregularity increases, the routing algorithm becomes more complex, as well as its cost and latency incurred at each node along the path. Thus, for very irregular and large topologies, routing tables represents a cheaper solution to achieve topology generality [PHKC09]. Besides, routing table compression techniques have been proposed [BCGK07] [PKH06].

Adaptive routing algorithms are more complex than deterministic ones, as they consider dynamic changing of the NoC state: balance the traffic on links, avoid congestion, etc.

Deterministic routing algorithms are simply and directly designed to be **deadlock-free** and **livelock-free**, according to their determined routes constancy. On the other hand, as all types of routes are possibly determined by adaptive routing algorithms, the deadlock and livelock freedom are ensured by respecting some routing restrictions and/or using supplementary virtual channels or networks.

Sometimes the characteristics of the path determined by a routing algorithm are considered relevant; thus, there are **minimal** and **non-minimal** path routing algorithms. Minimal path is usually the shortest one, so it normally incurs lower latency. However, this is not always the case, for example when congestions occur along the minimal path. In such cases, the path should be minimal from the latency point of view. Throughput-centric routing is addressed in [TDB03].

Taking into account **NoC area and power** consumption limitations, simple and low overhead 2D routing algorithms are preferred and have been designed for regular NoC topologies such as mesh, torus, spidergon, and ring [BC06] [CIB09]. In [MABDM06] a multi-path routing strategy is presented and evaluated. The method spreads the traffic in the NoC so that to minimize the bandwidth needs and power consumption.

In the embedded system domain, information about the applications that run on the system can be obtained, enabling therefore the design of **application-specific** routing algorithms [PHKC09]. Both guaranteed throughput and best effort services are taken into account in the router design presented in paper [RGR+03]. Moreover, the routing optimization for the system specifics can be considered early in the design [HGR05].

3.2.2 Fault-Tolerant Routing

Fault-tolerant methods are used to avoid both link and router failures. However, not all the faults that appear can be prevented or treated and they may produce failures. In paper [PNK+06], the internal structure of a router is considered and the impact of soft errors occurred in the different parts of the intra-router logic is analyzed. A special case that adaptability must deal with is when the NoC topology changes because of node and/or link failures. The routing adaptability in this situation is also referred to as *reconfiguration* [ZGT08] or *fault-tolerant routing*.

When failures of some nodes or links of the network are taken into account, even regular topologies manifest some degree of irregularity (unless spare nodes are used and the network reconfigures at lower levels). In such cases, even simple algorithms become more complex. Also, if routing tables are used, a supplementary overhead is incurred for updating them. However, the routing based on (possibly reconfigurable) routing tables [AN05] is always a possible fault-tolerant solution, even for regular topologies, if an optimized and specific approach is not available.

Fault-tolerant routing algorithms must not deteriorate network performances in the absence of faults. Usually, a simple (and possibly deterministic) routing is used in regular topologies in the absence of faults. After faults appear, adaptive (fault-tolerant) routing is used. Switching between deterministic and adaptive routing is also exploited to reduce congestions thus reducing the overall routing overhead [HM04].

Regarding the fault-tolerant routing, only mesh/torus topologies have been considered for NoCs [ZGT08] [FDC+09] [LLP09]. Even in macro-networks, fault-tolerant routing algorithms have been designed only for 2D mesh/torus topologies [CA95] [CB97] [CC01] [HS04], based on different approaches such as faulty blocks, turn models, intermediary nodes, virtual channels or networks etc.

Beside the above-mentioned approaches, another type of fault-tolerant communication has also been proposed, based on redundant messages in the NoC [DM03]. Fault-tolerant communication algorithms are analyzed in [PLB+04]: two different flooding algorithms (gossip and direct) and one random walk algorithm are investigated. The results of this analysis show that the flooding algorithms have an exceedingly high communication overhead. The redundant random walk overhead is significantly reduced, while useful levels of fault-tolerance and maintained. From the energy consumption point of view, the random walk is also considerably more efficient.

3.2.3 3D Routing

Work has also been done in the area of routing in 3D macro-networks. In [NGF+04] an adaptive and fault-tolerant routing for 3D meshes and tori massively parallel computing systems is presented. The method is based on intermediate nodes and implies storing at each node a table for routing, containing for each destination the list of intermediate nodes. When routing, the list of nodes is added in the message header. Once an intermediate node is reached, it is removed from the header. The method uses virtual networks and packets change virtual network at each intermediate node.

A fault-tolerant routing algorithm for 3D meshes, based on limited-global safety information is presented in [Wu03]. It is both adaptive and minimal. It uses the faulty cube model, built around faulty nodes and containing faulty and disabled nodes. Before sending a message, the method establishes a region of minimal paths. Regarding the 3D NoC topology with incomplete sets of vertical links and possibly different topologies in

horizontal layers, building faulty cubes would incur a very high number of non-faulty nodes that must be disabled.

Another fault-tolerant routing method for 3D meshes and tori is presented in [XSW05]. It uses the popular fault block model, but keeps it in 2D planes, where many *n*D unsafe nodes can be activated, enhancing thus the performance of the routing algorithm. Each fault-free node keeps its status in separate planes. Safety information of one node is about three times as much as that of the extended safety levels in [Wu03] in a 3D mesh/torus and *n* times in an *n*-dimensional mesh/torus. This is because each fault-free node needs to keep its safety information inside ((*n*-1)*n/2) 2D planes and the proposed safety measure in the whole system. Virtual sub-network partitions are used to avoid deadlocks. The number of virtual channels required by the method is linearly proportional to the number of dimensions of the network. 2D planes considered in this work are both horizontal and vertical and they have 2D mesh or torus topologies, which may not be the case of actual 3D NoC topologies. This fact makes the routing algorithm difficultly applicable to 3D NoCs.

In the paper [LPC05] a fault-tolerant routing algorithm for 3D torus is proposed. It can find a fault-free path between any two faulty nodes with a probability higher than 90% in a 3D torus with the number of faulty nodes up to 30%. Only local faulty information is used. The algorithm is based on rectangular boxes, which are not likely to be found all along the path in 3D NoCs, at least not at an acceptable price, paid as disabled non-faulty nodes count.

3.3 Coding and Recovery from Temporary Errors in NoC Transmissions

Codes are largely employed to *avoid*, *detect* and *correct* errors. The code must be selected depending on the fault type and the design of the protected system. Several fault-tolerant solutions based on data coding for NoC transmissions are briefly presented in this section.

3.3.1 Fault, Design and Cost Analysis

When designing a fault-tolerant scheme, the targeted faults (i.e. type, multiplicity, rate and place of occurrence) must be taken into account, as well as the costs they imply in terms of area overhead, latency, and power consumption.

In the SoC domain, *transient*, *intermittent* and *permanent* faults are all likely to occur and can lead to components or system failures. The majority of failures (80%) are caused by transient faults [LLP07], while the rest of them originate mainly in permanent and intermittent faults. Different techniques are employed to deal with permanent faults. For example, spare wires and cross points are used for yield improvement and self-repair (based on reconfiguration) to deal with life-time faults in NoC interconnects [GISP06]. TSV yield improvement for 3D NoC links is presented in [LML+08], based on spare via insertion. Serialization has also been proposed to improve the TSV yield [Pas09]. There are also techniques for protecting on-chip sequential elements against single event upsets [Nic05] [KCR06]. In this thesis we focus on tolerating transient faults in NoCs, based on coding and retransmission.

The fault *multiplicity* is important for choosing the appropriate code. In the NoC case, both *single* and *multiple* errors occur. Face to the high error rates of VDSM technologies, dealing with multiple errors becomes mandatory. Besides, multiple bit upsets

are likely to occur on adjacent bits. Faults are both *bidirectional* (‘0’ –> ‘1’ and ‘1’ –> ‘0’) and *unidirectional* (‘0’ –> ‘1’ or ‘1’ –> ‘0’) [BBDM05]. Faults that last more than one cycle (*multi-cycle*) are also likely to occur [YA08].

Faults occur in all types of components of a SoC. Their rate of occurrence depends on the design, technology, environment and operation conditions.

Fault-tolerant techniques based on coding are equally used in communication fabrics, computational cores and storage components. Regarding the NoC, codes (possibly combined with other fault-tolerant techniques) are employed to achieve fault-tolerance both in routers and links. Coding is applied to different types of components: transmission wires, logic circuits, storage elements etc. Depending on the router and link complexity, none, a few, or all types of components may present. The protected area consists of a single, as well as a group of such adjacent components in the communication pipeline. For example, a buffer that stores arriving data may be present in the design between the transmission wires and the switching logic (depending on the design modularity choice, such a buffer can be considered as part of the link or of the router). Different codes can be applied to protect the transmission wires and the buffer. Also, the same code can be applied to the wires and the buffer, if the code is able to deal with all types of errors that are likely to occur in these two components. Fault detection and location is addressed in [GIS+06]. The proposed mechanism uses code-disjoint for routing elements and parity check encoding for the inter-switch links to distinguish between faults in the communication links and faults in the NoC switches. Fault-tolerant router architectures have been proposed, by achieving protection of different router components [KNP+06] [FKCC06] [PNK+06]. However, in this work we do not focus on fault-tolerant architectures for NoC routers.

3.3.2 Avoidance, Detection and Correction Codes

Specific codes have been proposed to **avoid** specific error causes in the protected component. For example, many *crosstalk avoidance codes* (*CAC*) [PZGG06] [DZK08] exist for transmission wires and these codes are more efficient than other methods to avoid crosstalks (e.g. shielding [Mut07]). Also, codes that meet specific requirements have been designed, e.g. low-power codes. Joint CAC and error correction codes have also been proposed [PM04] [GPBG08]. A coding framework that implies several codes to address power, delay and reliability is presented in [SS04]. Crosstalk avoidance codes, low power codes and error detection and correction codes are jointly used in the proposed solution. In paper [RNKM05], Hamming and dual rail coding are considered for minimizing the crosstalk delays on on-chip busses. For less than 16 bits, DRC has lower power and crosstalk-induced bus delay, comparable with Hamming code.

Error detection codes (*EDC*) and *error correction codes* (*ECC*) are used to **detect** and **correct** errors whose occurrence could not be avoided. For multiple bidirectional error detection, the *Hamming* code is widely used [BBDM05] [SS04]. For burst and multiple error detection, *cyclic* codes, parallel blocks, and interleaving are used [BBDM05] [ZJ03]. The authors of [BTEF03] present an application of *m-of-n* codes for on-chip interconnects. Turbo codes are very popular for correction in communication systems, but they have a high implementation complexity and high latency. For the low latency and hardware overhead requirement of SoC designs, the use of Turbo codes is prohibited [BSCM06].

3.3.3 Recovery from Transient Faults

The main methods to recover from transient faults in NoC transmissions are *forward error correction* (*FEC*) and *error detection with retransmission* (*ED+R*). FEC is implemented by ECC, while ED+R makes use of EDC and retransmits data when errors are detected. These methods are applied either for each link in the NoC (*switch-to-switch S2S* or *link* protection) or only at network interfaces (*end-to-end E2E* or *transport level* protection). Regarding the flow control, the fault-tolerant techniques are applied either at each flit (*flit-based*) or to the entire packet (*packet-based*).

The retransmission is either *go-back-N* or *selective repeat*, depending on the receiver complexity [BBDM05]. When a packet (or flit) is erroneous, the receiver signals this fact to the sender. In the go-back-N scheme, the sender reacts by retransmitting the corrupted packet, as well as the following ones in the data flow. This way, the receiver does not have to store packets received out of order and reconstruct the original sequence. When the receiver has the capability of packet reordering, the corrupted packet is resent only (selective repeat).

In paper [BBDM02], the authors present power versus performance results for point-to-point error control in an on-chip protocol based on AMBA bus. The error correction and retransmission error recovery mechanisms are compared. The main finding of this work is that retransmission strategies are more effective than correction ones from an energy minimization point of view, in particular for long wires and high reliability constraints, because of the larger detection capability and the small decoding complexity. A similar analysis is performed in [BBDM05], this time for a NoC link and the conclusion is the same. However, the FEC overhead is expected to subside in emerging NoCs that span large designs using increasing number of hops and complex buffering/signaling structures. As the network size increases and, with it, the error rates increase, the energy and latency overheads of ED+R become unacceptable [BSCM06].

Different fault-tolerant schemes for NoC links are proposed in [LLP07]. They deal with transient, intermittent and permanent errors. The methods to deal with intermittent and permanent errors include spare wires with reconfiguration circuitry and time-information redundancy. For transient error protection, error coding (Hamming and interleaving) and retransmission are used. Both single and burst errors are considered for the fault-tolerance properties analysis. Handshake signals are protected using TMR.

An analysis of S2S and E2E data coding in NoC links is presented in paper [JLV05], from the power consumption point of view. The authors conclude that end-to-end error protection is more power efficient than both no protection and switch-to-switch protection. When voltage levels are reduced, signaling errors may occur, but they can be tolerated by using the error protection. This explains how error protected schemes can consume less energy than non protected ones.

From the transferred *information* point of view, not all data is equally critical. Specific codes can be designed to reduce the overhead by protecting the most critical data. An *unequal* error protection code against soft errors in packets is proposed in [DT07]. Critical parts of the packet such as packet header are protected more than data itself. The proposed code corrects all single errors and detects all double adjacent errors in the entire codeword. Additionally, it corrects all double adjacent errors in the header. The code is implemented for a router using *store-and-forward* routing. Unlike the *wormhole* routing strategy, where the flow is at flit level, in the store-and-forward approach, the entire packet is stored in the router buffer before it is forwarded to the next router or the

destination core. The overhead and latency incurred by the code are approximately as those of a single error correction code.

3.3.4 Hybrid and Optimized Error Control Schemes

Multiple power-performance-reliability trade-offs are explored in article [MBT+05]. Error control mechanisms at data link and network levels are investigated considering the energy efficiency, error protection efficiency and performance impact of different error recovery mechanisms. Static routing with paths set up at the sender NI and worm-hole flow control for data transfer are employed. Three classes of error recovery schemes are identified: end-to-end, switch-to-switch and hybrid. For the end-to-end detection scheme, parity and cyclic redundancy check at packet level are used. For the switch-to-switch error detection, the mechanism can be applied at flit or packet level. In the hybrid scheme, the receiver corrects any single error on a flit, but for multiple bit errors, it requests end-to-end retransmission of the data from the sender NI. The experiments show that for networks with long link lengths or hop counts, end-to-end detection schemes are power efficient. Switch-level detection mechanisms are power efficient when link lengths are small and when the end-to-end scheme requires large packet buffering at NIs. All schemes incur similar latencies for low error rates. At higher error rates, the hybrid error detection and correction mechanism provides better performance than the other schemes. ECC and EDC+R are also addressed in [NKRM05].

To reduce the energy wasted by error detection in favorable noise conditions, the redundancy of error detection schemes can be adapted based on the number of errors found over a fixed time window by a victim line operating at half supply voltage [LVKI03].

Low-density parity-check (LDPC)-based FECs are treated in paper [BSCM06]. LDPC codes are linear block codes suitable for low-latency, high gain and low power design. Error detection with retransmission, error correction and hybrid schemes are considered. The error detection with retransmission was implemented both for end-to-end and hop-to-hop communication types. The end-to-end scenario provides for better energy and latency performance at low error rates. The area overhead for the hop-to-hop scenario is four times larger when compared to the end-to-end case. For high error rates, the degradation in latency and energy becomes significant. In this case, the error correction with retransmission is more efficient: it is more energy efficient for long hop distances and has a reduced latency. The hybrid scheme uses the first scheme for shorter hop distances and the second for communication over long hop distances and with real-time communication constraints.

An adaptive error control scheme targeting multi-wire and multi-cycle errors is proposed in [YA08]. The scheme takes advantage of the adjacent error characteristics and configures a set of SEC codes and simple block interleaving to obtain different error correction capabilities. It is reported that the adaptive ECC scheme can achieve a 19% better energy efficiency than a fixed ECC scheme, while maintaining the same reliability level.

3.3.5 Adaptability to QoS

Several schemes have been proposed to achieve fault-tolerance while data is transmitted through the NoC. In order to obtain optimal efficiency, the impact on the application *QoS* level must be reduced, while the area, latency and power *cost* must be kept at minimum.

In [ZJ03], the authors present a fault model for NoC architectures, by analyzing the faults probability of occurrence, their characteristics and a distance metric influenced by the physical bus layout. The authors also propose different protection schemes for four QoS classes: *maximum bandwidth* (no encoding in this case), *guaranteed integrity* (error detection methods and retransmission are used), *minimum latency* (error correction) and *high reliability* (error correction and uncorrectable error detection are used, at the expense of lower bandwidth and higher latency). The error protection schemes are applied for switch-to-switch links.

The fault-tolerant schemes proposed in [VBC05] consider QoS for guaranteed throughput and best-effort traffic. One of the schemes implements single error detection using a parity bit and retransmission in the presence of error. The hardware overhead is negligible, but the latency per packet increases in case of retransmission. The other scheme implements single error correction using the Hamming code. A configurable error control scheme for NoC links that considers different QoS levels in terms of error control was also proposed in [RAM07].

3.4 Conclusions

- Checkpoint and Rollback

Different checkpoint and rollback protocols have been developed for macro-networks. The main checkpointing approaches are coordinated, uncoordinated checkpointing and communication induced. Different types of message logging (optimistic, pessimistic, causal) are used to complement the checkpointing, and the log is done either at the sender or receiver. Coordinated checkpointing is the approach the most preferred in practice because of its simplicity and low overhead. Scalability improvement has been addressed, by reducing the number of synchronization messages and that of checkpoints. Non-blocking protocols have also been proposed, as well as group-based checkpoint at common storage. Checkpoint portability approaches have also been addressed. These can be very useful for heterogeneous SoCs [LMS+05]. However, the proposed protocols treat equally all application tasks. In SoCs, communication patterns can be heterogeneous, and different sets of tasks can be run on the same SoC, independently or simultaneously. Therefore, QoS requirements must be met with minimum penalty incurred by the fault-tolerant mechanisms.

- Fault-Tolerant and 3D Routing

Many routing approaches have been proposed for 2D routing in distributed systems and NoC as well. A few routing algorithms also exist for 3D mesh and torus topologies. All these algorithms propose quite complex solutions, adapted rather for macro-networks, where overheads and latencies incurred are not as critical as for systems-on-chip. Besides, they all deal with regular 3D mesh or torus topologies, models that are not easily implemented for actual 3D NoCs, as shown in the previous chapter. 3D NoC topologies may have incomplete vertical connectivity and/or different topologies in 2D layers. Still, each 2D layer may have a regular topology. A possibility of using the above-mentioned 3D routing algorithms in 3D NoCs would be to model the 3D NoC irregularities as faults of the regular 3D mesh / torus topologies. However, considering the high irregularity of 3D NoCs, these algorithms are likely to substantially degrade their performances for these cases. General-topology routing algorithms are always a possible solution. However, routing algorithms for (partially) regular topologies are usually preferred over them, since generality implies higher costs in overhead and latency.

- Error Control Schemes

Codes are largely used in different components of the system (links and routers, in the case of NoCs) for tolerating temporary errors. They complement other techniques of improving the system dependability (e.g. spare wires and serialization to improve the wire yield). Recovery from transient errors in transmissions is achieved either by error correction or by error detection and retransmission. These mechanisms are applied at each link or end-to-end, at flit or packet level. Codes with high detection/correction capability can be implemented, but cost restrictions apply. Minimizing power consumption is one of the main requests that constrain fault-tolerant mechanisms, as a general characteristic for SoCs. Latency is critical especially for guaranteed throughput services. Hybrid and adaptive solutions have been proposed, considering different levels of fault-tolerance for specific QoS requirements of the application. However, these propose different types of codes and recovery mechanisms for each case. Dynamically switching among them implies high overhead, due to the presence of all these schemes in the design.

Chapter 4. Checkpoint and Rollback Recovery in NoC

This chapter presents application fault-tolerant protocols based on roll-back recovery concept. We evaluate the effectiveness and impact on NoC performance of the checkpoint and rollback recovery and propose enhancements to improve their scalability and to deal with different QoS requirements. The main classes of recovery algorithms are based on coordinated and uncoordinated checkpointing. Different solutions aiming to improve algorithm performance by exploiting the application partitioning and the blocking feature of the checkpoint are discussed. Scalability improving solutions at the protocol and routing levels are also proposed.

By disposing of a library of solutions, the most appropriate one in terms of performance/overheads can be selected for each application, depending on different factors such as application characteristics and QoS, and expected failure rate.

4.1 Checkpointing and Rollback Recovery

The *checkpoint and rollback* (*C&R*) mechanism allow the system to resume the execution after a failure from a later *consistent state* (i.e. *global checkpoint* or *recovery line*), prior to the failure. The rollback recovery concept compared with a failure-restart process is illustrated in Fig. 4-1.

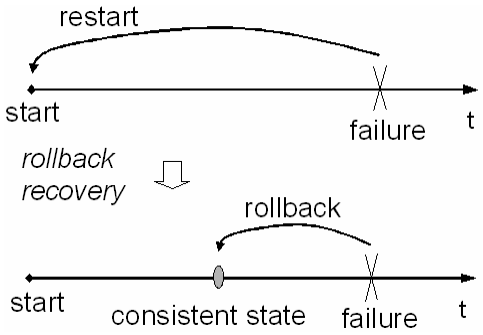


Fig. 4-1 Rollback recovery vs. restart

In the context of NoCs, the checkpoint and rollback technique is meant to deal with *transient* or *intermittent* faults (see chapter 2) that may provoke application failures, as well as with IP *permanent* failures. In the first case, the application execution can be resumed on the same IP, while in the latter it must be resumed on another IP. If proper serialization of checkpoints is employed, the failed and new IPs can have completely different architectures [ZP06] [DM05] [RS97].

The checkpointing mechanism implies periodical saving on *stable storage*, during error-free execution, of the information necessary and sufficient to reconstruct a *consistent state* in case of failure. The *stable storage* is a memory element whose content persists and is not corrupted by the tolerated failures. An application state is consistent if it could be reached following a correct and error-free execution of the application from the start point.

Before saving any state, the application must pass an *acceptance test* [Par96] [ASK08b], to avoid saving a state already contaminated by errors. Writing the acceptance test is the most critical part of the checkpoint and rollback mechanism and needs good knowledge of the application [ASK08a]. Because of its strong dependence on each respective application, the acceptance test is not detailed in this work.

4.1.1 Consistent State in Multi-Tasking Applications

The state of an application is represented by the states of all its components. The state of a task refers to the totality of information necessary and sufficient to resume task execution (context, registers, stack etc.). In the case of mono-task applications, this state represents the checkpoint. In a multi-processor architecture, the application is formed of several concurrent tasks mapped on the processors. The communication among tasks can be done by different mechanisms, such as shared-memories, message exchange or both. In this study, we focus on the case when tasks communicate by exchanging messages. However, the checkpoint-based recovery can also be implemented in shared-memory

communication. In this case, the shared-memory state consistency with the rest of the system must be managed [AG05] [SSC96].

A complete state of the multi-processor system consists in the totality of states of its elements (processing nodes, shared-memories, routers, links etc.). Different approaches can be implemented to save the state of the system. One of them would be to perfectly synchronize all system elements and save their state in the same time. To realize this, all elements must be able to save their states at a given moment. Another approach is to save only the states of computing nodes, as the first is not very realistic in complex systems. The latter approach is considered in this thesis. In this case, the *global checkpoint* of the system is formed of the (*individual* or *local*) *checkpoints* of its tasks. When a task saves its state as a part of the global state, we say that the task *take a checkpoint* or that the task *checkpoints*.

Even in the ideal case of the perfect synchronization of all tasks for checkpoint, there may be messages flowing in the network when the states of tasks on different processing nodes are saved. These messages can induce inconsistencies in the global state of the system. Four possible types of messages exist, as detailed in the following paragraphs and illustrated in Fig. 4-2. Messages are represented as arrows from their source to their destination task. Consider two tasks, denominated by T_A and T_B. Their individual states saved during the checkpointing phase are S_A and S_B, respectively. The global state of the application formed of T_A and T_B is composed of S_A and S_B. The dashed curve joining the two states depicts the recovery line.

- A message sent before its source checkpoint and received before its destination checkpoint is called *past* message. Past messages do not affect the consistency of the global state; they do not appear neither at the sender or at the receiver after the recovery line. *Message 1* in Fig. 4-2 is a past message.
- An *early (or orphan)* message emerges when the source checkpoints before sending it, while the destination task checkpoints after receiving it; as an example, see *message 2* in Fig. 4-2. The global state is inconsistent when early messages exist, as they appear as received, but never sent.
- When a source task checkpoints after sending a message and the destination task-checkpoints before receiving it, the respective message is called *late* (see *message 3* in Fig. 4-2). A system state with late messages is not consistent, as late messages appears as sent, but never received.
- A message that is send after the checkpoint of its sender and received after the checkpoint of the receiver is called *future* message (see *message 4* in Fig. 4-2). Future messages do not affect the consistency of the recovery line.

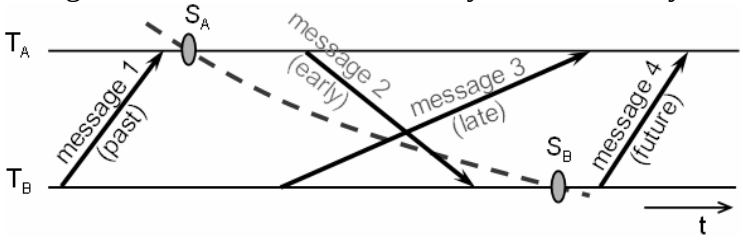


Fig. 4-2 Inconsistent global state with early and late messages

As a conclusion, early and late messages affect the consistency of the global checkpoint. On the visual plan, these messages intersect the recovery line (Fig. 4-2). If the number of tasks involved in the global checkpoint is high and the inter-task communication is dense, recovery lines without early and late messages may not exist, at least for certain laps of time. To *determine* (or *establish*) possible recovery lines, precise synchro-

nization of tasks and global view of the dynamic flow of messages in the network are necessary. To facilitate the existence of a relatively recent recovery line, *late messages are allowed* in our model. These are sent before the recovery line and are not normally resent after it, even if they are expected by their receivers. The solution consists in *saving* these *late messages* together with the checkpoint *and replaying* them after the recovery thus not affecting the global consistency. Accordingly, the information that a task saves as its checkpoint can include, beside the proper task state, a set of late messages.

In Fig. 4-3, a consistent state with late messages is presented. The same situation as in Fig. 4-2 is considered.

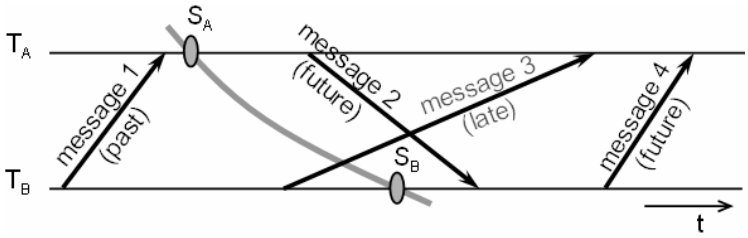


Fig. 4-3 Consistent state with late messages

Early messages must however be avoided. The reason can be found in their particular nature: they appear as received before the recovery line and sent after it (message 2 in Fig. 4-2). Originally, an early message was sent and received before the rollback. It remains received even after the rollback. As tasks resume their execution after the rollback, early messages are sent again (since their sending moment is after the recovery line). However, a task execution after recovery can be different than the execution before it. Therefore, the message content could also be different. Moreover, the message sending could not exist after the recovery. In addition, the global state of the system is already influenced by the old version of the early message, thus logging them is not a solution as in the case of late messages. Therefore, early messages are avoided in our model, by transforming them in future messages (see message 2 in Fig. 4-2 and Fig. 4-3, respectively). The transformation method will be detailed for each checkpoint and rollback approach.

4.1.2 Checkpointing and Message Logging Classifications

Several recovery approaches using checkpoints and rollback have been proposed. The difference consists in **when** and **how** the consistent state is determined. The main approaches are depicted in Fig. 4-4.

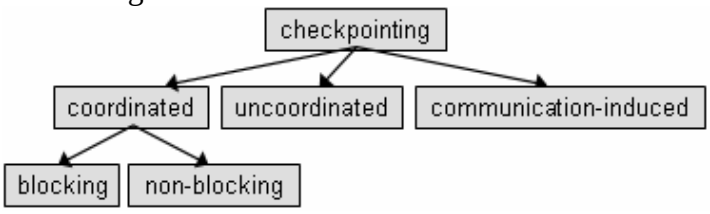


Fig. 4-4 Checkpoint and rollback classification

- In the *coordinated checkpointing* (CC) approach, tasks synchronize during the failure-free execution and save a consistent state on stable storage. In case of failures, they rollback to the last consistent state. Coordinated checkpointing is preferred in practice, given its simplicity and low overhead.

- In the *uncoordinated checkpointing* (UC) approach, tasks take checkpoints during the failure-free execution without any coordination. When a failure occurs, they synchronize in order to establish the most recent consistent state from the individual checkpoints; then, they rollback to this state. The main advantage is that no synchronization is needed to take the checkpoints, but it is subject to the domino effect [EAWJ02] in case of failure. The domino effect occurs when, starting from the most recent set of individual checkpoints of tasks, the recovery line regresses for one or several tasks at a time, to the previous individual checkpoint. The cause of this regression is the inconsistency of the recovery line. In turn, each regression can lead to new ones. This operation is repeated as long as a consistent state can not be established. In the worst case, the first consistent recovery line that could be established is identical to the starting point of the application. A solution to limit the domino effect is to complement the UC with message logging.
- The *communication-induced checkpointing* (CIC) is a hybrid method between coordinated and uncoordinated checkpointing, when tasks save individual states, but from time to time they synchronize to take a coordinated consistent state. However, it was shown that it does not scale well with a large number of tasks.

Message logging is meant to complement the checkpoint, as presented up to now. The main types of message logging are depicted in Fig. 4-5.

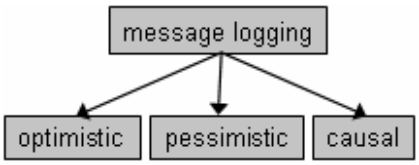


Fig. 4-5 Message logging classification

Message logging can be *optimistic* or *pessimistic*, depending whether the message influences the system before being logged on stable storage or not. The optimistic approach has lower overhead during the failure-free execution, as it avoids synchronous access to stable storage.

The *causal* message logging tries to combine the advantages of optimistic and pessimistic loggings, at the price of tracking the history of events that have influence on the state.

In this thesis, coordinated and uncoordinated checkpointing approaches are considered for implementation and analysis. Each of them is detailed in one of the following subsections. In our implementations, optimistic message logging complements both approaches, but it takes place in different phases for each of them, as it will be detailed in the respective sections.

4.1.3 Recovery Management Unit

A global synchronization phase of all communicating tasks is necessary for both uncoordinated and coordinated checkpointing. It is meant to establish a recovery line in the case of UC and to take the checkpoint in the case of CC.

Several protocols presented in literature stipulate that any task of the application can be the global *coordinator* of the global synchronization phase. Sometimes, the *initiator* of the global synchronization phase is also the coordinator. Several initiators can start different global synchronizations in the same time (or at least overlapped in time). In

such cases, supplementary communications in the NoC and local logic are necessary to elect a single coordinator and to take decisions at nodes when messages from different coordinators are received. Besides, since any task can be a coordinator, they all must contain the coordination module. The coordination logic can be quite complex, especially for establishing the recovery line in the case of UC. Unlike in macro-networks, not all PEs in NoCs are general enough to execute such algorithms beside their normal function. Often, they are dedicated to specific functions and are minimal, in order to reduce the chip area and unnecessary power dissipation.

To avoid the above-mentioned costs, we consider a single coordinator for a group of communicating tasks. However, any task can initiate global synchronization phases, by sending a message to the coordinator. In this work, we use a unique coordinator, called *recovery management unit* and denoted by *RMU*, both for UC and CC.

The RMU is a critical element to the checkpoint and rollback mechanism. In order to deal with failures that can affect it, the RMU has also a stable storage where critical information like the last recovery line is stored. As already assumed, errors do not affect the stable storage. Using this critical information on stable storage, a new instance of the RMU can continue the coordination task. The new RMU can determine if a synchronization phase was in progress. The other tasks are not concerned by the RMU failure if the latter occurred during an idle phase of the RMU.

If the RMU failure occurred during a synchronization phase, a solution to recover is to stop it and restart it later. In the case of CC, a new checkpoint will be taken. As for the UC, the recovery line will be recomputed.

The synchronization phase interrupted by a RMU failure must be stopped by informing the participating tasks; otherwise tasks could remained blocked in an intermediary phase of the global synchronization. Stopping a global synchronization phase can be done by the use of an exceptional *cancel* message to all tasks. The tasks must be prepared to treat such a command, even if they are not involved yet in a global synchronization phase, assuming that the failure of the RMU can occur any time. Other (complementary) solutions to deal with RMU failures are: spare copies and hardware/software redundancy and voting.

4.1.4 System and Application Models

The objective of this chapter is to evaluate the impact of different C&R protocols on the NoC performance, considering the overhead induced in the NoC to achieve global synchronizations of tasks. Therefore, the corresponding global synchronization protocols are developed. The tasks are seen as entities which send and receive messages. In this context, the fault-tolerance at application level consists in having all messages delivered at their respective destinations, even after eventual failures of one or several tasks. The assumptions regarding the application and system models are synthesized in the following paragraphs.

- *The application is formed of several tasks, each of them mapped on a node of the system. Tasks communicate by exchanging point-to-point messages. Each task is seen both as a traffic generator and as a traffic sink.* Regular application messages are sent from a source task to a destination task. The message is transferred either locally, if the two tasks execute on the same node, or using the NoC, if they run on different nodes. *No messages are lost (locally or in the NoC) and messages reach uncorrupted at their destinations, in a finite predetermined time.* These latter statements usually are based on other mechanisms at other levels of the communi-

cation protocol stack, such as: *communication failures do not partition the network* [SS94], *the routing algorithm can find a path between any pair of nodes*, and *lost or corrupted messages are retransmitted or corrected.* These assumptions will be guaranteed at transport, routing and data link levels by providing appropriate fault-tolerant mechanisms, as described in chapters 5 and 6.

- *Each task is provided with mechanisms for on-line fault/error/failure detection* [ACT00] [JS04] [LYMC07] [ZLKK07]. *The fault detection mechanism checks that the current state is error-free and the messages received are error free.* This mechanism ensures that all checkpoints are error-free (i.e. the *acceptance test*). Otherwise, if a checkpoint contaminated by an error is used for recovery, the failure will occur again after the application execution is resumed, thus the recovery is compromised. If a *task failure* occurs, the task will stop execution and no longer send or receive messages (*fail-stop* or *fail-silent* mode). Data in task volatile memory is lost. Only the content of the stable storage is not corrupted and can be retrieved after the failure. *Upon the failure of a task or node, a failure message is automatically sent to the recovery management unit (RMU).* This message is sent by a module of the respective node, not affected by the failure. If the node does not have such modules, a possible option is represented by the node neighbors. In this case, we assume that any non-faulty node is aware of the failures of its neighbors (e.g. by exchanging "*I'm alive*" messages [AN05]).
- *Each task has a stable storage memory, which is not affected neither by the failure of the task nor by the failure of the node.* Information stored in it can be retrieved even after the above-mentioned failures. Several methods of implementing a safe stable storage can be used. Some examples can be found in [CMH96] [CKS01]. *The stable storage is local to each task and reading and writing operations are atomic and instant.* This assumption is unrealistic, considering the time to transport the data to/from the stable memory and the one to read/write it. Nevertheless, these operations can be considered instant if they can be done in parallel with the application task; otherwise they can be back-annotated with timing information.
- *Each task can save its state at any moment.*

4.2 Uncoordinated Checkpointing and Rollback Recovery

In this subsection, we present the uncoordinated checkpointing and rollback we developed for the application model presented above. It is based on the classical uncoordinated checkpointing, complemented with message logging to reduce the domino effect.

The principle of the uncoordinated checkpointing is illustrated in Fig. 4-6. It consists in saving the task state independently by each task and recovering from failure in a centralized recovery phase. A global synchronization is also necessary to delete older and no longer necessary checkpoints (garbage collection), as explained subsequently.

Each task can take several checkpoints before a rollback is necessary and the memory occupied to store a checkpoint can be large, especially for big message logs. With several checkpoints, the stable storage must be considerably large or it will be overflowed. On the other hand, as newer checkpoints are taken, older ones can no longer be useful to construct a consistent recovery line. Therefore, a garbage collection mechanism for recycling the recovery memory is necessary. The garbage collector we developed is

centralized and non-blocking, allowing thus a small latency overhead during the failure-free execution of the application.

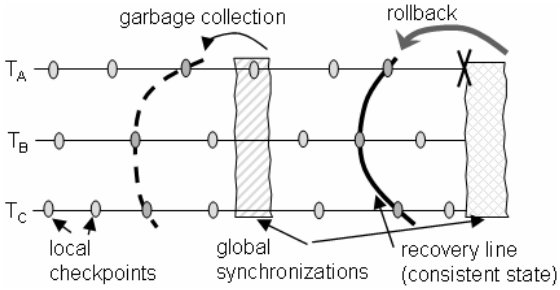


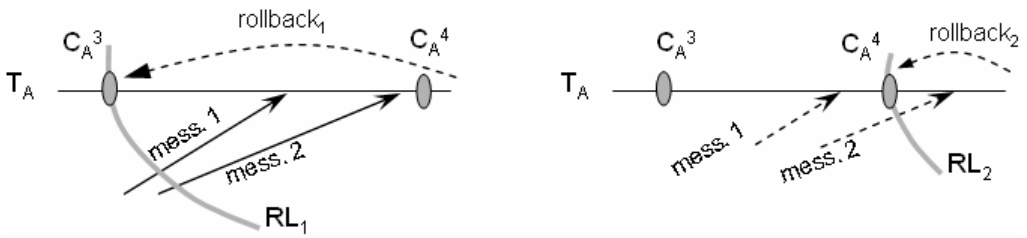
Fig. 4-6 Uncoordinated checkpointing principle

The message logging is performed at the receiver, in an asynchronous manner, by keeping a volatile log, which is periodically flushed on stable storage. The approach is optimistic, since it assumes that the log is saved to stable storage before a failure occurs. This reduces the latency during the failure-free execution, but must be treated during recovery. Logging the messages at receiver (vs. sender) incurs smaller latency and network overhead when replaying them, after eventual rollbacks.

4.2.1 Local Checkpointing

We use an optimistic approach to log messages. When a task takes a checkpoint, the information is at the beginning stored in a volatile memory. Thus, until the moment it is flushed in the stable storage, it can be completed with small latency. When task T_A saves on stable storage its checkpoint C_A^i, it saves: T_A task state and the message log. Messages received since the previous checkpoint, C_A^{i-1}, are considered. Also, messages already in the current replay list must be stored along with the checkpoint, as explained in the following paragraphs.

A task may need to replay a set of messages (the late ones) after a rollback. If this entire set is not replayed before the next checkpoint, the remaining messages must be saved on stable storage during the next checkpoint. This ensures the continuity from one checkpoint to the next one and, in the mean time, the independency of the next checkpoint. An example is presented in Fig. 4-7.



a. a first rollback b. a second rollback

Fig. 4-7 List of late messages at each checkpoint

After a first rollback to C_A^3 (see Fig. 4-7.a), messages 1 and 2, stored in checkpoint C_A^4, are identified as late. A list of late messages is created. In our case, it is formed of these two messages.

Observation: After the rollback, all past checkpoints (relative to the recovery line) are erased. All future checkpoints are processed, in order to identify all late messages; then, these

checkpoints are erased, too (see the subsection treating the rollback). Therefore, messages 1 and 2 are not necessarily received before C_A^4.

The task continues its execution and at proper moments, it replays its late messages (see Fig. 4-7.b). After being replayed, a message is deleted from the list. Suppose the new C_A^4 is taken after message 1 was replayed, but before the replay of message 2. After a second rollback (to the new C_A^4), the message 2 must be replayed. Therefore, the list of late messages must also be stored in the checkpoint. In our case, it contains only message 2. After the rollback, the list will be completed with all other messages that will be identified as late.

The task state is saved for restoring the task context after eventual rollbacks. The log contains the messages, which are saved for future replays, in case the established recovery line identifies them as late messages. Both of these elements will be used locally, to recover the respective task. They are not useful in determining the recovery line.

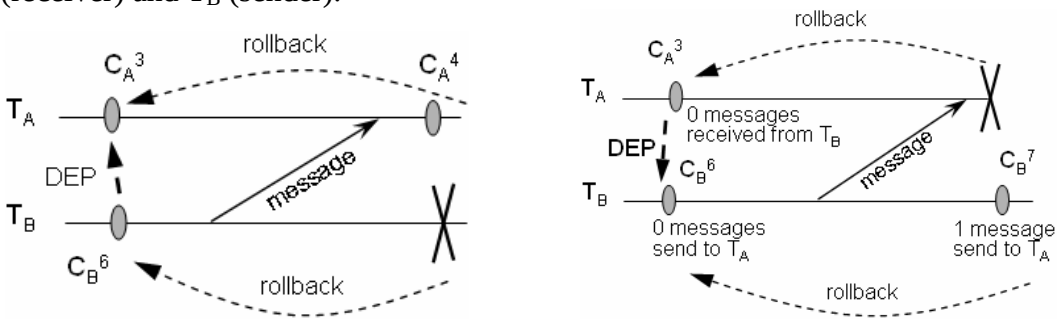
To determine the recovery line, each task sends to the RMU the state consistency information (SCI) they possess. This is formed by the following elements:

- The list of the available checkpoints;
- For each checkpoint, the additional information that must be saved in the log. This contains the following elements:
  - A list of checkpoint IDs of other tasks. Checkpoints in this list are the last ones taken by the sender of the received messages since the previous checkpoint, C_A^{i-1};
  - The number of sent and received messages to/from all other tasks.

The additional information and its role are detailed in the rest of this subsection. As presented earlier, it is used at RMU to establish a consistent recovery line. This is realized by determining dependencies between different checkpoints, based on this additional information. A *dependency* between two checkpoints means that a rollback to one of them implies the rollback to the other one. They can be unidirectional or bidirectional. Another role of the additional information is after the rollback, to offer support for identifying the late messages.

Index of the sender last checkpoint: Avoid orphan messages and identify late ones

During the recovery phase, a consistent global state must be determined using information from all tasks. Orphan messages (received, but not sent) must be avoided. This is done by forcing a rollback of the receiver to a state previous to the receipt of the otherwise orphan message. This transforms the orphan message in a future one. An example of such a situation is depicted in Fig. 4-8.a. A message is exchanged between T_A (receiver) and T_B (sender).



a. Dependency induced by the failure of the sender b. Dependency induced by the failure of the receiver

Fig. 4-8 Different types of dependencies

The most recent checkpoints of the two tasks are C_A^4 and C_B^6. However, a recovery line formed of these two checkpoints classifies the message as an orphan one. To avoid

this, T_A must rollback to a checkpoint prior to the receipt of the message. The most recent checkpoint that respects this condition is C_A^3. The condition appears as a restriction when establishing a consistent recovery line: a rollback of T_B to C_B^6 forces always a rollback of T_A to C_A^3. This results in a dependency between C_B^6 and C_A^3, labeled by $C_B^6 \rightarrow C_A^3$, and depicted in figures (see Fig. 4-8.a) by a dashed bold arrow (DEP). This dependency will be used by the RMU to establish the recovery line. The dependency is determined at the receiver. Upon the receipt of each message, the receiver establishes a dependency between the last checkpoint of the sender (when the message was sent) and its last checkpoint (in the moment of reception). For this, all messages piggyback the index of the last checkpoint of their sender. This is a first item of the additional information to be stored in the checkpoint as part of the message log. This information is also useful to identify late messages, as it will be detailed in the subsection treating the rollback.

Avoid late messages loss

In our implementation, the message log is stored at the receiver. However, in the case of optimistic logging, the receiver is subject to failure before accomplishing the transfer of the last log on stable storage. In this case, after recovery, the eventual late messages that are not on stable storage are lost. An example is presented in Fig. 4-8.b. The receiver (T_A) fails before saving on stable storage its next checkpoint (that is C_A^4). The most recent recovery line that can be established is formed by checkpoints C_A^3 and C_B^7. After the rollback, the message appears as late, but it does not exist in the log, therefore it can not be replayed. The message is lost. A solution to this situation is to force the sender to rollback to the checkpoint prior to the message sending, in our case, C_B^6.

To realize this, some information related to the existence of the message must also be tracked in another checkpoint related to the message. That is a checkpoint of the sender. However, saving all sent messages at sender induces a high overhead and redundancy (the messages are logged both at the sender and the receiver). Our solution consists in tracking at each task only the number of sent and received messages to/from all other tasks. At recovery, these numbers are used in determining additional dependencies for avoiding the message loss due to the receiver failure. In the example in Fig. 4-8.b, the number of messages received by T_A from T_B is 0, as recorder in C_A^3. The checkpoint where the number of messages sent from T_B to T_A is also 0, is C_B^6. A dependency is therefore created between C_A^3 and C_B^6. Thus, a rollback of T_A to C_A^3 forces a rollback of T_B to C_B^6, transforming in future message the otherwise lost message.

4.2.2 Synchronization Protocol for Recovery and Garbage Collection

Checkpoints are memory intensive. However, checkpointing should take place at shorter intervals than expected application failures (expressed in MTBF), in order to ensure the progress of the application execution. In our scheme, when a rollback is performed, all checkpoints, both previous to and succeeding the recovery line, are removed. At such moments, the amount of required memory overhead can be reduced. However, memory allocated for checkpoints can grow significantly if failures do not occur regularly. Moreover, a part of these checkpoints may no longer be useful, as a consistent recovery line can be constructed by newer checkpoints.

The garbage collection determines the most recent consistent recovery line and deletes all checkpoints preceding it. This is done similarly to a recovery procedure, except

that there is no failed task and the start-up criterion is the amount of memory occupied by checkpoints.

Therefore, a recovery line is necessary both for recovery after failures and for garbage collection. The recovery / garbage collection synchronization to determine a recovery line for the uncoordinated checkpointing protocol is depicted in Fig. 4-9.

The RMU sends a *state consistency information* request (SCI_REQ) message to all application tasks. Consequently, they send their *state consistency information* (SCI). After receiving all SCIs, the RMU builds a recovery graph. Based on the recovery graph, the RMU establishes a recovery line and broadcasts it (REC_LINE) to the tasks.

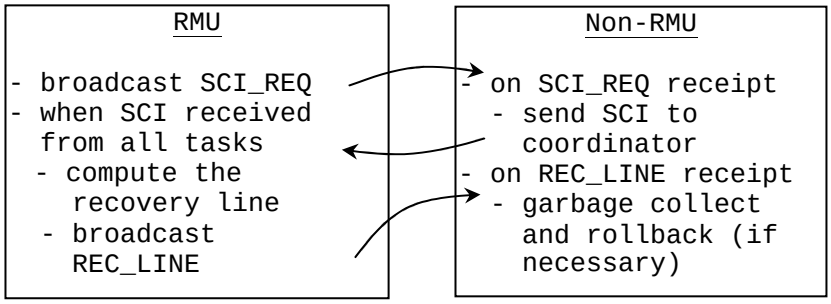


Fig. 4-9 Uncoordinated recovery protocol

Observation: The term *broadcast* is used identify the situation when a message is sent from a specific task to all others.

The rollback procedure in the case of uncoordinated checkpointing is more complex than in the case of coordinated checkpointing. The recovery line indicates for each task the checkpoint to resume execution from. It also indicates its type: a rollback recovery line or a garbage collection one.

In case of a **rollback** recovery line receipt, the task must rollback to the indicated checkpoint. The message log associated to this checkpoint and the previous checkpoints can be removed. The following checkpoints message log must be processed considering the recovery line: all late messages from these logs are added to the replay list of the recovery line checkpoint (these include the eventual replay lists of these checkpoints). Then, the following checkpoints can also be removed. The task resumes execution from the recovery line checkpoint state, and, beside the regular application messages, messages from the replay list are also replayed. If a new checkpoint is taken before the replay list is exhausted, the remaining messages are added to the replay list of the new checkpoint, so that in case of garbage collection or rollback, the previous checkpoint could be safely deleted, as detailed in the next subsection.

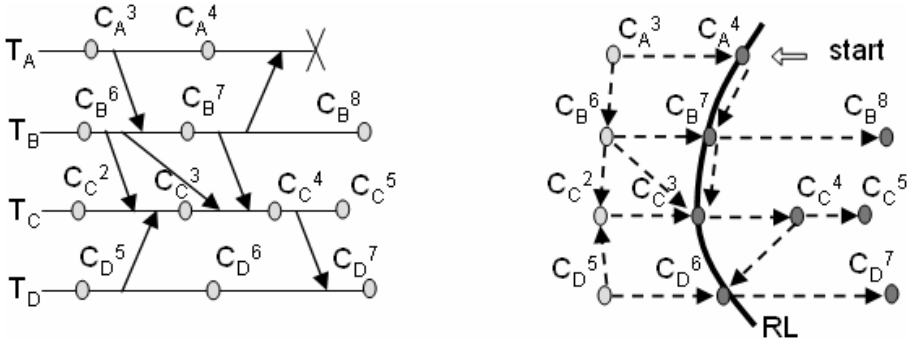
In case of a **garbage collection** recovery line receipt, there is no need to rollback the task. The checkpoint in the recovery line corresponding to the current task is part of the most recent possible recovery line. Therefore, the checkpoints before this checkpoint are removed (together with the message log of this checkpoint).

4.2.3 Recovery Line and Rollback

Upon receiving state consistency information from all tasks, the RMU is able to determine the global recovery line. The recovery line is computed using a recovery graph built from the consistency information. The recovery graph is a directed graph. The nodes are the task checkpoints and the edges result from dependencies. Beside the two types of dependencies presented in the precedent subsection, a dependency is considered

from each checkpoint of a task to the next one. The reason for adding this dependency in the graph is to direct the recovery line towards the most recent checkpoints.

The recovery line is determined by traversing the recovery graph in a recursive manner following all possible edges from the current node to unvisited nodes, marking each traversed node as *visited*. The oldest visited checkpoint of each task will be part of the recovery line. An example of how a recovery line is established is depicted in Fig. 4-10.



a. Failure scenario b. Dependency graph and recovery line

Fig. 4-10 Establishing a recovery line

In the scenario presented in Fig. 4-10.a, messages are represented as arrows. In Fig. 4-10.b, the dashed arrows represent the dependencies. For example, a dependency induced by a message receipt is C_A^3 –> C_B^6. The dependency C_A^4 –> C_B^7 is created from the number of send/received messages between tasks T_B and T_A. The reversed dependency, C_B^7 –> C_A^4, induced by the last message from T_B to T_A was lost because of the failure of T_A. The entry point of the algorithm is given by the last available checkpoint of the failed task, C_A^4 in this case. Then checkpoints: C_B^7, C_B^8, C_C^3, C_C^4, C_C^5, C_D^6 and C_D^7 are visited (the order is not important, it depends on the specific implementation of the algorithm). The recovery line is formed of: C_A^4, C_B^7, C_C^3 and C_D^6, i.e. the oldest checkpoints visited for each task. After the recovery line is thus established, the RMU broadcasts it to all involved tasks.

Upon receiving the recovery line, each task performs rollback to its corresponding recovery checkpoint. At this moment, late messages are identified. Each message received after the recovery checkpoint is processed. If its source rollbacks to a checkpoint placed after the message was sent, the message is scheduled for replay. The task execution is resumed from the recovery checkpoint. In Fig. 4-11, the rollback is illustrated for the same example considered earlier for determining the recovery line. Task T_C has to schedule for replay the late message sent by task T_B, as T_B will not resend it.

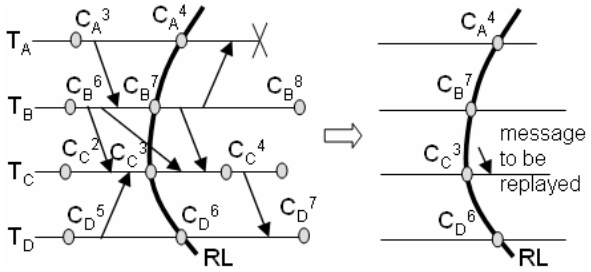


Fig. 4-11 Rollback

Another operation executed upon rollback is the removal of all checkpoints, except those belonging to the recovery one. Thereby, if a failure occurs before a new recovery

line can be determined using new checkpoints, this last recovery line is still available. The removal of all other checkpoints is permitted, since they are useless. The checkpoints after the recovery line can be deleted without any loss, since tasks resume execution and new checkpoints (maybe different) will be generated. The checkpoints before the recovery line can also be deleted, as it is not possible to form a future consistent recovery line which would contain checkpoints older than those in the current recovery line. This is a direct consequence of the recovery line establishing method, which considers the oldest checkpoints that can be reached by recovery graph dependencies. New dependencies from future to past are impossible to occur, since all orphan and late messages are already treated.

4.3 Coordinated Checkpointing and Rollback Recovery

In this section we present the coordinated checkpointing protocol developed and optimized for our NoC application model.

4.3.1 Principle of Coordinated Checkpointing

The goal of the coordinated checkpoint is to execute a global, consistent checkpoint. Thus, when recovery is performed, all tasks resume from the latest global checkpoint. The coordinated protocol we developed requires little extra information to be stored by each task and does not use time markers, therefore it can be used both for synchronous and globally asynchronous, locally synchronous (GALS) NoCs [CCCM06].

In the CC approach, the application tasks execute normally and from time to time they perform synchronization in order to save a new consistent global state, as illustrated in Fig. 4-12. The interval between two successive checkpoints is called “epoch”. In the case of CC, unlike the UC, each individual checkpoint belongs to a recovery line, thus the epoch is also the interval between two successive possible recovery lines.

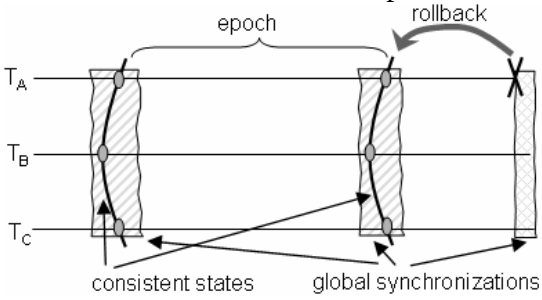


Fig. 4-12 Coordinated checkpointing principle

After a new recovery line is established, it can be used for resuming the application execution in case of failure. The previous recovery line can be removed. Thus, at any moment, at most two checkpoints must be kept on stable storage for each task. Usually, after the coordinator is informed by all tasks that the new local checkpoint is taken, it replies with a broadcast message that validates the new recovery line.

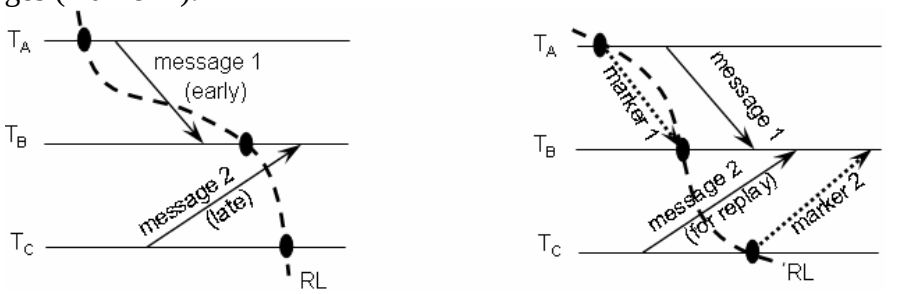
In our protocol, any task can initiate a checkpoint or a rollback, but the coordination of both checkpointing and rollback is done by the same dedicated coordinator, the *RMU* (as stated in the RMU section). Having a unique possible coordinator has several advantages. One of them is that the validation broadcast can be skipped. Instead, a task knows that its last checkpoint is valid either upon receipt of a new checkpointing phase or upon rollback, since in both cases the RMU communicates the index of the respective check-

point. Besides, the unique coordinator approach avoids the synchronization needed when several initiators start a checkpointing phase, thus reducing the associated overload in the communication network. Moreover, passing rollback requests by the unique coordinator avoids the situation when, in case of rollback request from another task, some processes have already received from the former initiator the validation of the latest checkpoint and others have not.

As there may be messages flowing in the network when different processes take their local checkpoints, the fault-tolerant protocol must handle early and late messages in order to obtain a consistent global checkpoint.

In order to avoid early messages, if each message piggybacks the index of the last checkpoint of its source, upon receiving the message, the destination can checkpoint before processing it. Thus, the first message received from a future epoch forces the destination to take a checkpoint. Besides, this message (that piggybacks the index of the last checkpoint of its source) can also serve for marking the end of the list of late messages. Thus, the first message sent after the new checkpoint of its source can end the list of late message received by the destination from this source. The main inconvenient of the piggybacking solution is that the first message after checkpoint can be produced a long time after the checkpoint was taken. This induces a significant prolongation of the checkpoint phase. As each task must be informed on every other task (about the end of its checkpointing phase), the above-mentioned situation has a significant occurrence probability. To avoid very long checkpoint durations induced by the piggybacking method, we choose the marker solution to deal both with early messages and late message list ending. It consists in informing all other tasks upon taking a new checkpoint.

On the left side of Fig. 4-13, an inconsistent global state is depicted, with message 1 being early and message 2 being late. Markers can be seen on the right side of Fig. 4-13, depicted as dotted arrows. They serve either to avoid early messages (marker 1) or to log late messages (marker 2).



a. Inconsistent state b. Consistent state with late messages

Fig. 4-13 Using markers to obtain consistent global states

A task is considered to have taken its checkpoint only after both its state and the complete list of the late messages received are on stable storage. Late messages are to be replayed after an eventual rollback. In practice, the task cannot effectively take its checkpoint before knowing that there are no more late messages, i.e. all tasks begun the new epoch.

Two types of markers have been proposed in earlier works: piggybacking the current epoch index on the application messages and sending a dedicated message to all other tasks.

The acceptable arrival delay of the piggybacked marker (NEW_EPOCH) is conditioned by the communication pattern. In order to speed-up the checkpointing protocol, in our implementation, dedicated marker messages supplement epoch piggybacked markers.

4.3.2 Checkpointing and Recovery Protocols

The **checkpointing** synchronization protocol executed by tasks in order to establish a consistent recovery line is presented in Fig. 4-14.

The RMU broadcasts a checkpoint request, CK_REQ, to the application tasks. Then, these exchange markers (CK_START) in order to establish a consistent state. After its local checkpoint has been taken, each task informs the RMU about this fact, by sending a CK_TAKEN message. Then the RMU can validate the global checkpoint, which is formed of the local checkpoints.

When a task knows its new local checkpoint is globally validated, it can remove its previous checkpoint from stable storage; thus, at most two checkpoints must be stored at any moment.

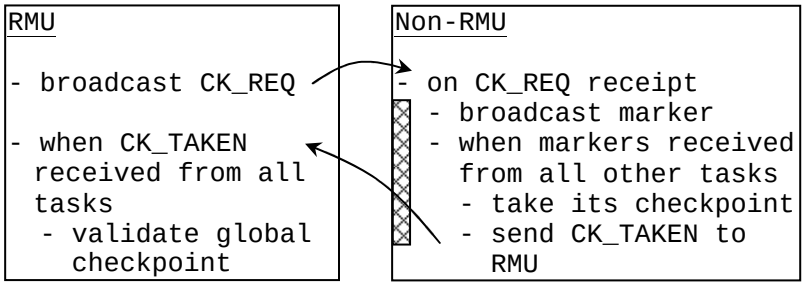


Fig. 4-14 Coordinated checkpoint protocol

The coordinated checkpointing can be executed either in **blocking** (B) or **non-blocking** (NB) approach, depending if the task blocks its normal execution during the checkpointing protocol or not. More details are given in subsection 4.5.2.

The **rollback protocol** is coordinated by the RMU. Actually, when a failure is detected, the RMU broadcasts a rollback message to all tasks. The index of the checkpoint to rollback to is contained in this message. Each task will resume the execution from the designated checkpoint.

Failures must also be dealt with during checkpointing and recovery [AHW05], besides the normal operation. Failures of both RMU and non-RMU tasks are tolerated, independently of their moment of occurrence:

- Non-RMU task failures
  - During normal operation. In this case, a rollback to the previous checkpoint is performed. Therefore, the failure is discarded.
  - During checkpointing (before CK_TAKEN is sent). In this case, the new checkpoint can not be validated and it is discarded. A rollback to the previous checkpoint is performed.
  - During rollback. The rollback procedure is re-operated to the same checkpoint.
- RMU task failures. In this case, the RMU function is migrated to a spare RMU, using the stable storage of the failed RMU.
  - During normal execution of application tasks. This failure does not influence the application tasks.
  - During checkpointing (before the new checkpoint can be validated). In this case, the new checkpoint is discarded.
  - During rollback. The rollback procedure is re-operated by the new RMU, to the same checkpoint.

4.4 Uncoordinated and Coordinated Checkpointing Features

This subsection synthesizes the common characteristics of the uncoordinated and coordinated checkpointing protocols that we have developed and adapted for NoCs, briefly pointing the benefits of the choices that were made, as well as the differences between the two approaches.

The **similarities** of uncoordinated and coordinated checkpointing are:

- A unique RMU is used, thus avoiding coordinator election phases and related synchronizations;
- The checkpoint is combined with the message logging, thus facilitating the existence of recent recovery lines;
- Messages are logged at the receiver; i.e. replaying them does not induce traffic load on the NoC;
- Consistent states with late messages but no orphans are considered;
- Messages piggyback the index of the last checkpoint of their sender;
- The log is done in an optimistic manner, reducing the fault-free execution overhead;
- Multiple failures are tolerated. They can occur anytime, even in the checkpointing synchronization phase of the coordinated checkpointing.

The **differences** between the uncoordinated and the coordinated checkpointing protocols are presented in Table 4-1.

Table 4-1 Uncoordinated vs. coordinated checkpointing

Differences	Uncoordinated Checkpointing	Coordinated Checkpointing
logged messages	***all***	***late messages***
necessary checkpoints	***at least 2***	***at most 2***

All messages are logged in the case of uncoordinated checkpointing, as in the moment of logging, there is no information about which of the messages are late. In the case of coordinated checkpointing, only late messages are logged, as these can be identified at the moment of log.

The number of necessary checkpoints in the case of uncoordinated checkpointing is at least two: one belonging to the most recent recovery line and the new one that is taken. Several checkpoints can be taken until a new recovery line is established and older checkpoints can be deleted. Having a very recent recovery line is important, as the goal is to reduce to the minimum the amount of processing to be redone after the failure. However, having a recent recovery line implies frequent checkpoints. On the other hand, the checkpointing implies an overhead in the normal execution of the application. Also, having a lot of checkpoints implies frequent garbage collection phases, which also induces latency. Considering these facts, the checkpoint frequency must be upper limited. On the other extreme, if the checkpoint frequency is too small when compared to the failure frequency, the recovery line can not advance between two successive failures. Therefore, the checkpoint frequency must be set according to the failure frequency.

In the case of coordinated checkpointing, at most two checkpoints are necessary, the one already validated and the new one. Once the new one is validated, the previous one can be removed.

4.5 Checkpointing Adaptability to QoS and Scalability Improvements

4.5.1 Coordinated Checkpointing with Reduced Number of Broadcasts

Considering its simplicity in synchronizing a global, consistent fault-free state, coordinated checkpointing is preferable in practice. However, with the increasing number of PEs, the coordinated checkpointing suffers from poor scalability [EP04]. This is because global synchronization required to build a global consistent state induces a significant communication overhead for large systems, which increases the checkpoint latency (the time the protocol takes to save a new checkpoint, including the global synchronization). The checkpoint period is lower-bounded by the checkpoint latency. Therefore, larger checkpoint latency forces larger checkpoint periods, which result in ampler rollbacks, and consequently in larger recovery times (as the execution interval from the checkpoint is wasted), impacting the performance of the system. Moreover, if the synchronization duration stretches more than the MTTF, there is no time to take new global checkpoints between two successive failures. In such cases, rollbacks are performed to the same old checkpoint and thus, the application execution does not advance. In the case of coordinated checkpointing, the duration of the global synchronization is also called (global) checkpoint duration. Therefore, in the case of coordinated checkpointing, not only the overhead induced by taking local checkpoints must be reduced, as in the case of uncoordinated checkpointing, but also the duration of the global synchronization, as local checkpoints have no sense alone in this case (i.e. CC).

Our objective is to improve the scalability of the checkpoint recovery mechanism by reducing the duration of the global synchronization and thus the checkpoint latency.

The acceptable arrival delay of the piggybacked marker is conditioned by the communication pattern. In order to speed-up the checkpointing protocol, a second type of marker supplements the epoch piggybacking, but it is not done by messages sent from all-to-all tasks, which can significantly overload the network traffic. Instead, it results from a centralized communication phase with the RMU: all tasks inform the RMU about their new epoch (CK_START) then, in turn, the RMU broadcasts the marker message (CK_START_ALL) to all tasks. Thus, the marker overload is reduced from $O(n^2)$ to $O(n)$, where n is the number of application processes. The two types of markers have the same role and can be used interchangeably, whichever arrives first. The one that arrives second is ignored. In Fig. 4-15, the coordinated checkpointing protocol developed in this work is presented.

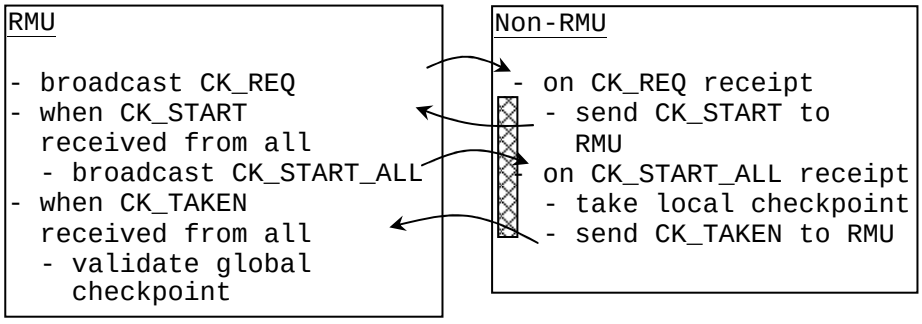


Fig. 4-15 Coordinated checkpoint with reduced number of broadcasts

The RMU broadcasts a checkpoint request CK_REQ (Fig. 4-15). As soon as possible after receiving it, each application task saves its state and sends the marker of its new epoch, CK_START, to all other tasks. It continues its execution until markers from all other tasks are received. When all markers are received, it saves its checkpoint on stable storage and informs the RMU that its checkpoint has been taken by sending CK_TAKEN. When the RMU receives CK_TAKEN from all tasks, it can validate the global checkpoint. Normally, the validation is done by broadcasting a message to all tasks. We consider that a checkpoint is validated upon the receipt of the CK_REQ for the next checkpointing phase, in order to avoid the explicit validation broadcast. As soon as a task knows its new local checkpoint is globally validated, it can remove its previous checkpoint from stable storage.

When the classical coordinated checkpoint protocol (Fig. 4-14) is used to take a global checkpoint, at least n broadcasts are executed, where n represents the number of PEs in the system, as explained in subsection 4.3.2. The broadcasts significantly loading the communication fabric are the CK_STARTs that are sent among tasks, introducing an overhead of approximately n^2 messages in the network. This number becomes significant when the number of tasks increases: 12 messages for a 4x4 mesh NoC, but 65280 messages for a NoC built on a 16x16 mesh! Besides, sending these messages is a bursty process, and will tend to create congestion, consequently increasing the checkpointing latency.

In this checkpointing protocol, the all-to-all broadcast among tasks is replaced by a single message transmission per application task (CK_START sent to RMU by all tasks) and a broadcast (CK_START_ALL). The number of synchronization messages is thus reduced from $O(n^2)$ to $O(n)$. The associated rollback protocol is unchanged.

4.5.2 Blocking and Non-blocking Coordinated Checkpointing

Both the basic coordinated checkpointing protocol and the reduced number of broadcasts version can be executed with blocking on not of the task normal execution.

In the non-blocking (NB) approach, tasks continue normal execution during the checkpointing phase; synchronization messages are sent and received together with application messages. The same sequence of actions is executed in both approaches, except that the task execution is blocked between the CK_REQ receipt and the CK_TAKEN sending, as indicated in Fig. 4-14 and Fig. 4-15, by the lattice block. However, during the checkpointing, late messages are logged and will be part of the checkpoint. This choice (instead of waiting for network channels to be empty) reduces the checkpointing duration. Since the blocking and non-blocking protocols are the same, each task can participate to the global checkpointing either by blocking or not its normal execution, regardless of the other tasks. A consistent global state will be checkpointed, independently of the configuration of blocking and non-blocking tasks.

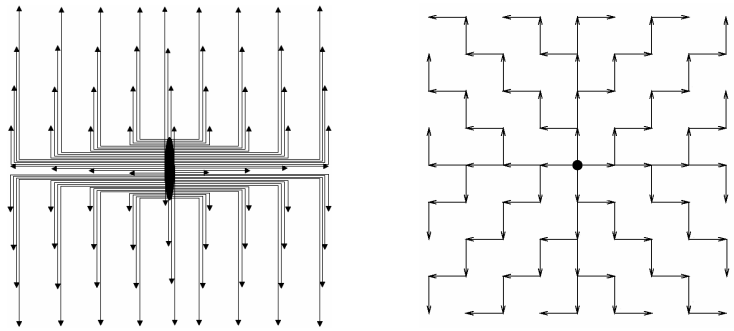
The decision of blocking or not its normal execution can be taken for each task and for each global checkpoint, depending on the application QoS requirements and the actual traffic load in the NoC. For example, if the execution of a task is critical to the application or real-time and should not be interrupted during the global checkpoint, it will participate at the checkpointing without blocking its execution. On the other hand, if a task incurs a high traffic load but its execution is not critical, it will be blocked. This is due to the fact that the checkpointing period can be stretched exactly by the high traffic on the NoC, thus blocking the tasks inducing such traffic can contribute in reducing it.

In our model, the B-NB decision of a task is independent of the other task choices, since tasks can save their state at any moment. Without this possibility, other restrictions on B-NB choice must be considered, such as common decision for groups of communicating tasks.

4.5.3 Smart Broadcast

In subsection 4.5.1, we have shown that when the number of broadcasts is reduced in the CC synchronization protocol, the scalability is improved. However, a single broadcast can induce congestion in the network, if classical point-to-point routing is used. This subsection proposes the use of a "*smart broadcast*" for the global synchronization among tasks. We consider mesh topologies to illustrate it, but similar reasoning can be applied for other topologies.

Classical algorithm of broadcast to *n* nodes injects *n* messages in the network. As there are no unique links among all different PEs, messages sent by a broadcast follow some common links before reaching to their respective destinations. Thus, a node closer to the source of broadcast must pass several messages to further nodes, while their contents are identical. Fig. 4-16.a illustrates the loading of links in a classical broadcast scenario from the central node in a 9x9 mesh using static XY routing. The horizontal links on the same line with the sender node have to bear a very high traffic load, especially the links closer to the sender. It can be observed in the figure that the load on the links is highly unbalanced.



a. Classical broadcast by XY routing b. Smart broadcast

Fig. 4-16 Classical vs. smart broadcast

An example of one-to-all efficient broadcast for mesh networks is illustrated in Fig. 4-16.b, for the same 9x9 mesh. This efficient broadcast is based on a simple algorithm [YW99] that implies multiplying a broadcasted message on the fly. The source node sends the message to be broadcasted only to its first-order neighbors. Then, each node provides the message to its associated PE and forwards copies of it to its first-order neighbors, in a way that each node receives the message only once. The traffic load on the links is balanced and minimum.

The simulation results will show the effectiveness of this technique applied to the coordinated checkpointing when compared to a classical XY broadcasting. Nevertheless, it can also be used by uncoordinated checkpointing.

4.5.4 Partition Configurations

Applications running on MPSoC platforms can be partitioned in several communicating groups. Also, several distinct applications can run in the same time in an MPSoC. Thus,

we can consider separate checkpointing for different partitions of the application(s). Thus, the critical characteristics of the fault-tolerant protocols can be improved, i.e. the time to determine the recovery line (in the case of UC) and the global synchronization duration (in the case of CC) can be reduced. We propose two optimized recovery configurations for the coordinated checkpointing, which can also be adapted and applied to the uncoordinated checkpointing approach.

The optimized recovery configurations take advantage of the NoC operation phases when the communication pattern is partitioned. Thus, in order to reduce the checkpoint latency, each task could take its checkpoint in coordination with the tasks in its own partition only. In Fig. 4-17, the two optimized RMU configurations are depicted: *shared RMU* and *multiple RMU*. The nodes running tasks in the same partition are represented with the same texture (in this example we suppose that if several tasks are running on the same processing element, they belong to the same application partition).

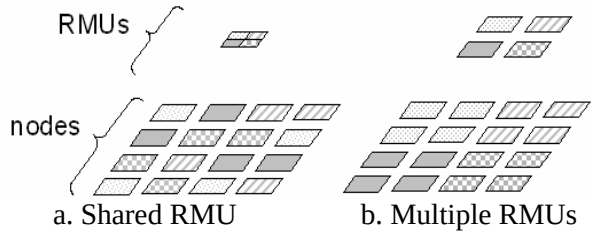


Fig. 4-17 Optimized RMU configurations

For the shared case (Fig. 4-17.a), the same RMU is used by all partitions in all their intra-partition synchronizations. The shared RMU is more complex that the *simple RMU*, as it has to be able to manage several partition synchronizations. Symbolically, it is depicted as being formed of several parts, one for each partition. However, partition synchronizations are not treated serially (i.e. a new partition synchronization does not have to wait the end of the current one), as this RMU reduces such latency overheads by allowing overlapping synchronizations of different partitions.

In the multiple RMU configuration (Fig. 4-17.b), each partition has a dedicated RMU for intra-partition synchronizations. Each RMU is represented with the same texture as the corresponding partition. In this configuration, RMUs only need to handle synchronizations in a single partition; they act like simple RMUs for their respective partition. This solution requires higher costs, as several RMUs must be available in the NoC. However, this configuration allows better performance when the communicating tasks are localized, as each RMU can be located close to its partition, or, ideally, in the central point of the partition. Thus, the communication latency between tasks and their corresponding RMU is reduced.

In order to suggest the localization advantage of the multiple RMUs, tasks in the same partition are depicted as localized in Fig. 4-17.b and not localized in Fig. 4-17.a. However, both configurations can be used whether the partitions are localized or not.

4.6 Experimental Results

4.6.1 NoC Simulator and Application

4.6.1.1 NoC Simulator

The NoC modeled for simulations is a homogeneous infrastructure built of routers and links. Functional cores communicating through the NoC are denoted here by *IP* (*Intellec-*

tual Property) cores. Routers and nodes can be connected together through links to build NoCs of **arbitrary topologies and sizes** (irregular or regular: mesh, torus, tree-based etc.).

The **switching protocol** that governs the data transmission is **wormhole**, in which packets are divided into basic flow control units (**flits**). The first flit of each packet, called the *header* flit, reserves the routing resources (buffers and links) along the path between a source and a destination. The data flits, carrying the useful information, follow the path reserved by the header, while the tail flit (the last flit of the packet) ends the transmission and frees the resources reserved by the header.

The **simulation abstraction level** is timed-TLM (transaction level model) [CG03].

The NoC simulator has the ability to work with various **traffic** sources: traces files collected from applications running on real systems or generated synthetically off-line.

Safety features are also implemented, such as: deadlock detection, topology consistency checking (check whether unconnected NoC elements exist), routing consistency checker (i.e. check whether a route between a source and a destination exists before sending a message).

The structure of the **simulation environment** is presented in Fig. 4-18. The core of the simulator is the simulation engine, written in Java, that simulates the NoC elements which process the messages at flit level, implement routing policies and flow control etc., and the simulator also collects measurement data.

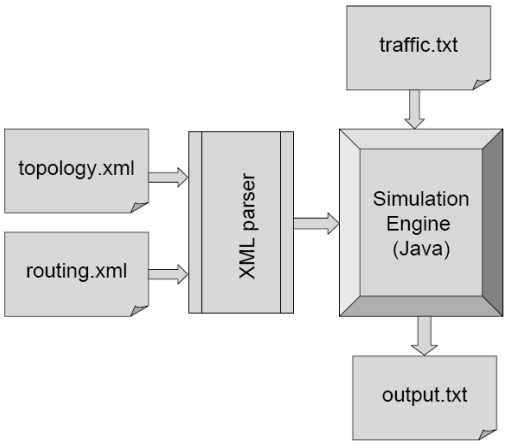


Fig. 4-18 Block diagram of the NoC simulator

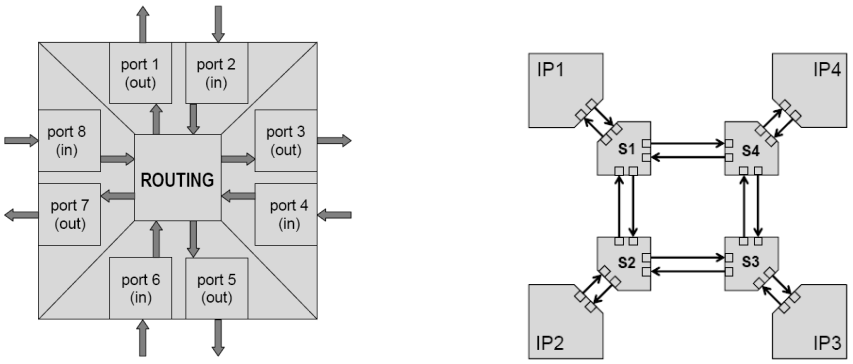
Two of the simulator inputs refer to NoC topology and routing policy. This information is contained in two different input files (*topology.xml* and *routing.xml*). These files can be generated automatically for regular, hierarchical and ad-hoc topologies, and semi-automatically for irregular topologies (see Fig. 2-2 for examples of topologies). The relevant data is extracted from these files by an XML parser and fed to the simulation engine. Another input of the simulator is the NoC input traffic information, defined in text format in file *traffic.txt*, which is read during simulation. Data packets are created by the simulator according to the traffic description in this file. Finally, status information and measurement data are written in the *output.txt* file.

Routers, links and IPs are each modeled by a dedicated Java class; their main characteristics are presented hereafter.

The NoC **switch** model (*networkswitch* Java class) is based on a three-stage pipeline corresponding to an input/output buffering scheme, as shown in Fig. 4-19.a. The three stages correspond to input buffering, routing/arbitration, and output buffering operations,

respectively. In cases where the NoC uses switches with less or more than three stages, the switch model can be modified accordingly by editing the *networkswitch* class.

Switches and IPs exchange data through *ports* (as depicted in Fig. 4-19.b, which are unidirectional in the current implementation. The number of ports depends on the particular NoC topology and is defined in *topology.xml* input file.



a. NoC switch model b. IPs, switches and links connected in a NoC

Fig. 4-19 NoC switch model and NoC based system example

Each switch has a number of virtual channels that can be set by the user. The capability of each virtual channel (in terms of number of flits it can store) can also be set by the user.

Each NoC **link** connects two router ports or a router port and an IP port. In the current implementation, all links are unidirectional. For bidirectional communication, a pair of links in opposite directions must be instantiated. The latency (in number of cycles) along the link can be set by the user. The default link latency value is one.

The NoC-based system's functional nodes, the **IPs**, are modeled with respect to their communication interfaces (Fig. 4-20). They serve two main purposes:

- Extract messages from the traffic input file (*traffic.txt*) and place them into their internal communication queues, whose size is user-configurable.
- Inject/extract data flits to/from their output/input ports.

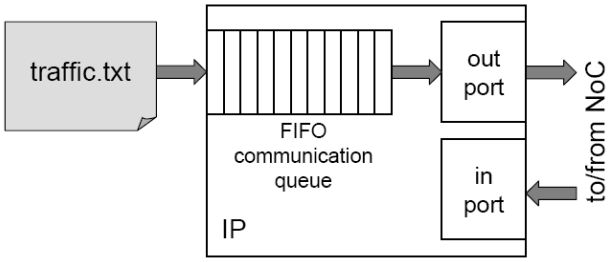


Fig. 4-20 IP model and traffic injection

Messages are moved from the traffic input file into the IP internal communication queue in one simulated clock cycle. From there, data moves into the output port of the IP, one flit per simulated cycle (if free space exists in the IP output buffers).

4.6.1.2 Application and C&R Implementation

The **application** consists in several tasks; each task is seen both as a traffic generator and a traffic sink. Considering the traffic generator functionality, each task injects messages

of a given length, with a given frequency. The destination of the message is usually randomly chosen among the other tasks. With these parameters, uniform traffic patterns are generated. The traffic sink part consists in the receipt of the messages destinated to the respective task. In this application model, no dependencies are considered between the sending of a message and the receipt of other messages. Similarly, there are no dependencies between the receipts of messages from different sources. Considering this application model, the role of the NoC is to ensure that all messages injected reach their intended destinations.

The **input traffic** is implemented using the file that contains all messages injected in the NoC by all tasks, in a chronological order (*traffic.txt*). We call this file *input file*. The messages that are received represent the **output traffic** and are flushed in an *output file*. A *reader* from the input file is implemented for each task. After the rollback, the reader repositions to the next message after the rollback checkpoint. In order to validate the correct operation of the NoC (i.e. all messages reach to their intended destinations), the input traffic must be identical with the output traffic, i.e. all messages sent must be received in the same order they were sent, between each source-destination pair. After the simulation is completed, the input and the output file are analyzed and compared, as explained hereafter. In the output file, only the messages from the same source are ordered; message from different sources can have a different order than in the input traffic file. For the comparison of input and output files, messages from each source to each destination, in chronological order, are identified and written in an intermediary file. Such an intermediary file is created for both the input and the output traffic files. Then, considering the same order for the source-destination pairs, the two intermediary files must be identical if all messages injected reached their respective destinations.

The pointer of each task in the input file keeps track of the sent messages. Because of eventual rollbacks, a message can be sent several times. However, at the receiver, only the most recent version of the message must be considered. In the following paragraphs we detail the implementation of this feature for both the coordinated and the uncoordinated checkpoint approaches.

- In the case of the **coordinated checkpointing**, a list of messages received since the previous checkpoint is maintained at each node. When a global checkpoint is taken, the list of the received messages can be flushed in the output file. Indeed, if a failure occurs from this moment on, the task will rollback to this checkpoint and thus, the received messages will not be received in the future. This happens because their sending moment is before the checkpoint of the sender (otherwise they would be early messages) and the sender will also never rollback at any point before this checkpoint.
- In the case of the **uncoordinated checkpoint**, a list of messages received since the previous local checkpoint is also kept for each task. However, they can be flushed in the output file only when they are certainly before a virtual recovery line. This can be established after computing a recovery line, either after a failure or upon a garbage collection process. More exactly, the list of received messages since the previous checkpoint can be flushed on the output file when the checkpoint is deleted or it belongs to a recovery line.

After **rollback**, each task must resume its execution from the recovery checkpoint; i.e. it resumes its message injection starting with the next message after the last one sent on the checkpointing moment. Concerning the future messages sent before the rollback took place, they are discarded.

The checkpoint and rollback protocols we developed are independent of the **mapping** of the application tasks on PE. Thus, several tasks can be executed on the same

node. However, all simulations consider the same mapping: one-task-on-each-PE. This mapping is considered to evaluate the worst case of the protocol impact on the NoC performance. This way, the any-to-any task communication involves the NoC for any pair of tasks. Besides, tasks sharing PEs are penalized in their communication with tasks on other PEs, as the access to the NoC is also shared. Moreover, the local managing overhead of several tasks on the same PE can be ignored.

Messages from a source to a destination must be delivered to the destination in the same order they were sent. This requirement is observed by the XY routing, as all messages from one PE to another follow the same route. However, to comply also with routing policies that do not respect the in-order delivery of messages, messages from a source to a destination are indexed and buffers for **message reordering** are implemented at the receiver.

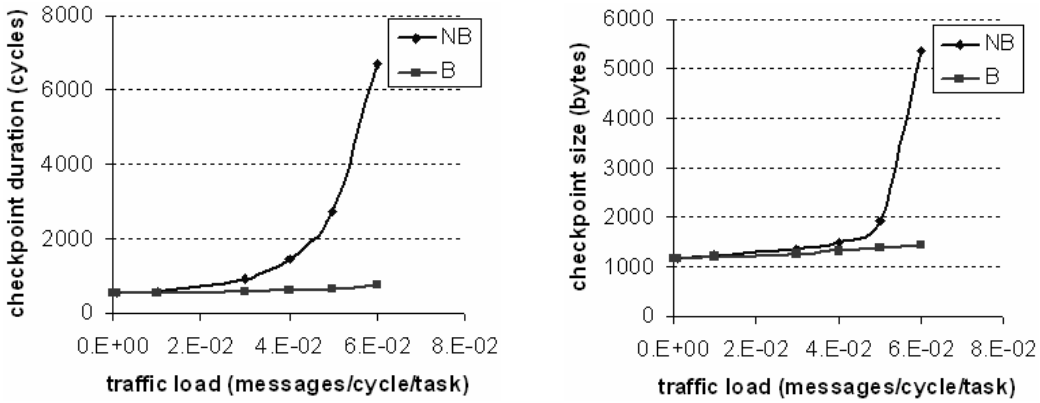
4.6.2 Simulation Results

The simulator presented in subsection 4.6.1 is used for the simulations presented in this chapter. The NoC topologies used are direct-connected meshes and the routing policy is XY. A task is mapped on each PE and the RMU task shares a PE with one of the application tasks. The traffic injected is uniform; it is measured in messages/cycle/task and each message has a constant length of 64 bytes. However, different message granularity could also be considered. In this subsection, the terms *message* and *packet* are used interchangeably, as each message is formed of a single packet, to reduce the exploration space.

4.6.2.1 Blocking and Non-blocking Protocol

This section presents the results obtained by simulation using the blocking and non-blocking coordinated checkpointing protocol, under different traffic loads and failure rates. All simulations are run on a 4x4 mesh direct-connected NoC.

The first results evaluate the average checkpoint duration and overhead (Fig. 4-21) for the blocking and non-blocking approaches, considering different uniform traffic loads, from a very light one to one approaching the maximum throughput of the NoC.



a. Checkpoint duration (cycles) b. Checkpoint overhead (bytes)

Fig. 4-21 Checkpoint duration and overhead for B-NB CC

We observe from Fig. 4-21.a that for very low traffic loads, the checkpoint durations of the blocking and non-blocking approaches are approximately the same. However, as the traffic load increases, the checkpoint duration in the non-blocking case significantly

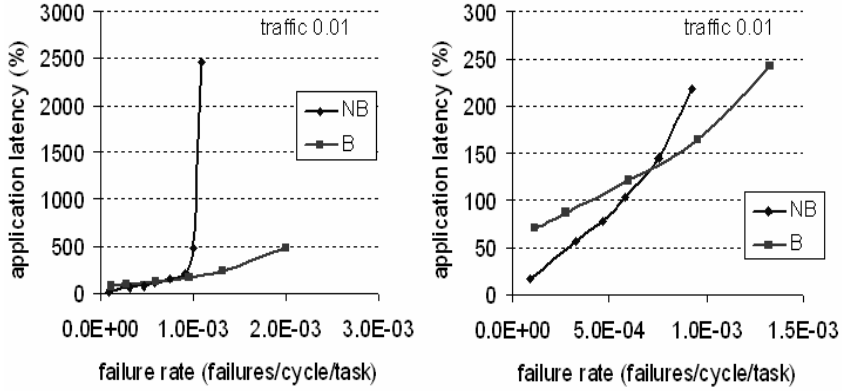
increases (the diamond-marked curve). In the blocking case (the square-marked curve), the checkpoint duration presents only a slight increase. This is explained by the existence or non-existence of the application traffic in the NoC. In the non-blocking case, the recovery traffic is significantly delayed by the application traffic. In the blocking case, as the application traffic is stopped, recovery traffic is not significantly delayed. However, the amount of application messages that flow through the NoC when the blocking starts is proportional to the traffic load, which explains the slight checkpoint duration increase with the traffic load in the blocking approach.

Fig. 4-21.b depicts the checkpoint overhead in the same scenarios. The same significant ascending trend with the traffic load increase can be noticed for the non-blocking approach.

As the traffic load and the checkpoint duration increase, the probability of late messages becomes more significant, which explains the increase of the checkpoint overhead. On the other hand, the blocking approach keeps the overhead at very reasonable limits when compared to lower traffic load situations.

The checkpoint overhead analyzed here refers to the amount of memory occupied by a checkpoint of a task, measured in bytes. It includes the task state and the message log. The state of each task depends on the task itself and the way it is implemented (at the OS level or at the application level). When done at the OS level, the state takes more space, as all the necessary information is saved, but does not require the programmer intervention. When done at the application level, the programmer is aware of the checkpointing mechanism and can establish points where the checkpoint requires smaller memory to be stored. In our estimation we considered a fixed amount of the task state for all tasks (140 bytes in our simulations). The message log size is computed by summing up the sizes of the messages actually logged and the sizes of the information necessary for their management (message id, size, source, arrival time, order number etc.).

The next experiment studies the latency induced in the application execution. The latency performance is evaluated relative to the ideal case when there are no failures and no checkpoints are taken. Two traffic loads are considered: 0.01 and 0.03 messages/cycle/task (Fig. 4-22 and Fig. 4-23, respectively) and different failure rates.



a. application latency (%) for B-NB b. zoom on the B-NB intersection point

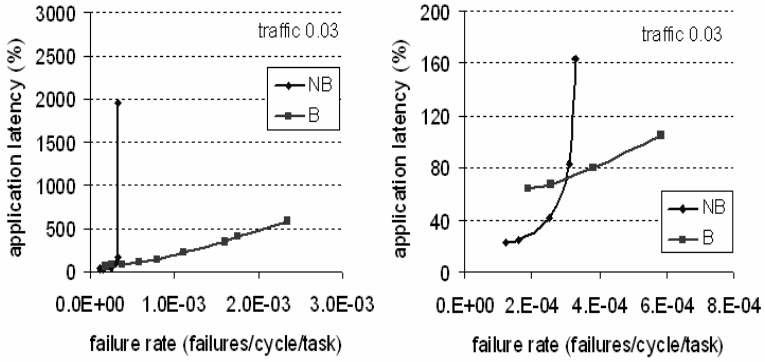
Fig. 4-22 Application execution latency for B-NB CC - traffic load of 0.01 (messages/cycle/task)

Normally, the application execution is delayed in the blocking approach as the application is blocked during checkpointing phases. This can be observed for relatively low failure rates (the failure rate is expressed in number of failures/cycle/task). However, for higher failure rates, we can observe (Fig. 4-22.b, the intersection point zoom) that the

blocking approach induces smaller latency than the non-blocking one (intersection point at 7E-4 failures/cycle/task).

Moreover, the non-blocking approach becomes ineffective for higher failure rates, which is not the case for the blocking one. In fact, as the checkpointing duration is larger in the non-blocking case than in the blocking one, the time interval between two successive failures is not long enough to take a new non-blocking checkpoint, for higher failure rates. Thus, in the non-blocking approach, the probability to rollback to the same old checkpoint increases. In consequnce, parts of the application are re-executed several times, which leads to extremely high latency penalty.

For a three times higher traffic load (Fig. 4-23.a and b), the same trend is maintained, but the intersection point between the two approaches occurs for a lower value of the failure rate (3E-4 – see right side of Fig. 4-23).



a. application latency (%) for B-NB b. zoom on the B-NB intersection point

Fig. 4-23 Application execution latency for B-NB CC - traffic load of 0.03 (messages/cycle/task)

This is due to the fact that the difference between the checkpointing duration of the blocking and non-blocking approaches is larger for higher traffic loads (as seen in Fig. 4-21).

Thus, if the expected failure rate is between the two traffic intersection points, different approaches (blocking/non-blocking) are preferable for different traffic loads.

4.6.2.2 Scalability Improvements

This section presents simulation results that illustrate two scalability improvements applied to the coordinated checkpointing: one at the protocol level, aiming at reducing the number of broadcasts, and the smart broadcast.

The XY routing algorithm is used as a baseline, while the smart broadcast presented in section 4.5.2 is used as the improved broadcast. We used for simulations uniform traffics with a constant rate of message injection for every task (0.005/cycle).

Several mesh sizes (4, 16, 64 and 256 nodes) for the NoC system were simulated, as illustrated in Fig. 4-24.

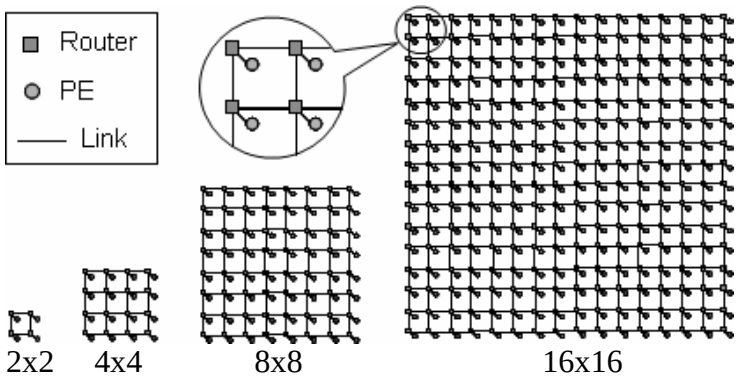


Fig. 4-24 Different sizes of NoC meshes used for simulations

Simulation results are presented in Fig. 4-25 and Fig. 4-26. The checkpoint latency for the four different sizes for the NoC-based system is presented in Fig. 4-25.

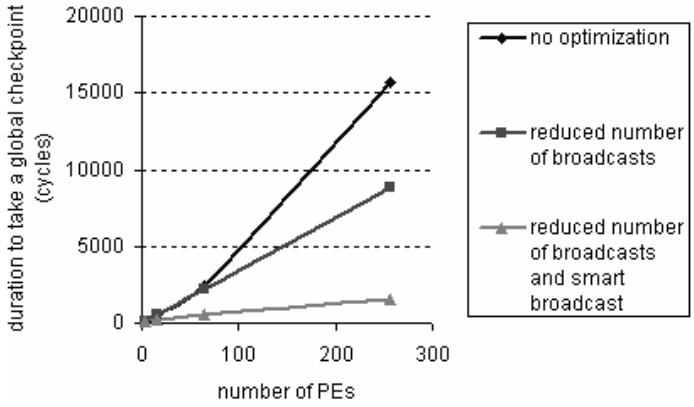


Fig. 4-25 Scalability improvement of global checkpoint duration

The diamond-marked curve represents the checkpoint latency for the checkpointing protocol with no optimization, using a classic XY broadcasting. The square-marked curve represents the case when the number of broadcasts is reduced in favor of the more centralized synchronization. The checkpoint latency when the smart broadcast method is applied is depicted by the triangle-marked curve. We note that the overhead of executing a checkpoint can be significantly reduced (with up to one order of magnitude in case of large systems) when less synchronization messages are required to implement the checkpointing protocol and the broadcasts are optimized.

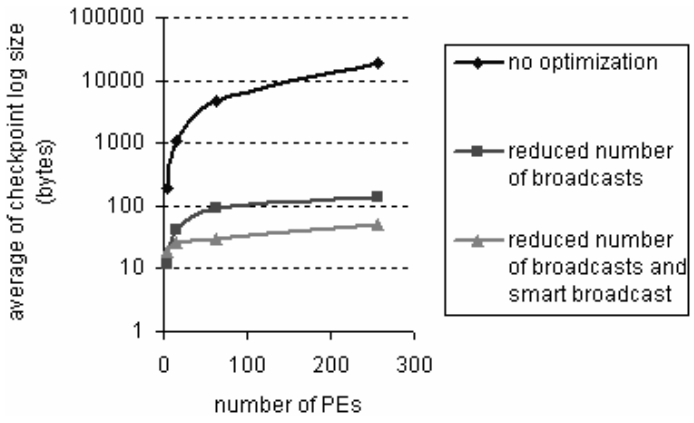


Fig. 4-26 Scalability improvement of local checkpoint message log size (bytes)

The size of memory occupied by the log of late messages was also measured. In Fig. 4-26 the message log size for a task is presented, for the same three checkpointing method implementations as those used for Fig. 4-25. An important reduction of several orders of magnitude of the log memory can be observed in Fig. 4-26 when using the optimized implementations. As the checkpoint latency decreases, the probability to receive a message from the previous epoch also decreases. This explains the decrease of log size with the decrease of the checkpoint latency.

Simulation results indicate that the relative improvement in terms of latency and memory overhead becomes more significant as the number of PEs increases, effectively enhancing the scalability of the recovery mechanism.

4.6.2.3 Uncoordinated vs. Coordinated Checkpointing

The number of checkpoints stored on stable storage at any moment is higher in the UC than in the CC protocol. This is because CC saves on stable storage up to two local checkpoints, while UC needs at least two checkpoints, in order to ensure the advance of the recovery line.

With the following simulations, we perform a comparison between the two recovery methods considering the necessary memory for a single checkpoint (the task state and the corresponding message log and dependencies). The average checkpoint memory overhead was measured using three traffic loads, as shown in Fig. 4-27. The simulations were run on a 4x4 direct NoC.

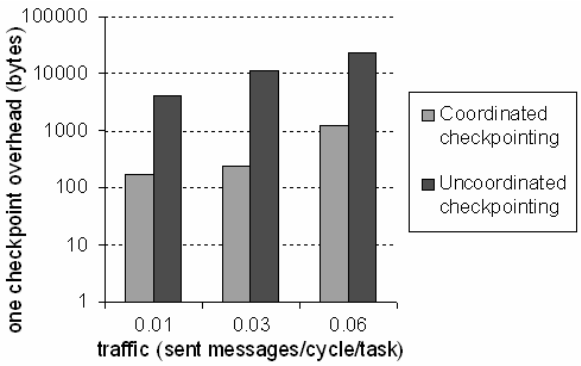


Fig. 4-27 CC vs. UC: Memory overhead per checkpoint

Considering the same application (having the same traffic pattern), UC needs more memory than CC. UC logs during a checkpoint all messages received by the task since the previous checkpoint, while the CC approach logs only late messages. As the task state is constant for the two methods, the size of a checkpoint depends on the message log size.

As it can be seen in Fig. 4-27, the size of a checkpoint also increases with the traffic. Since the checkpointing frequency is the same for all cases, this increase is directly determined by the increase with the traffic of the number of received messages, and respectively, of late messages (for the coordinated checkpointing).

The coordinated checkpointing synchronization is performed in the same way, independently of the traffic. Nevertheless, higher traffic loads can lead to an increase of the checkpointing period, as it can be seen in Fig. 4-28. We also measured the checkpointing period of the CC using all-to-all markers. When compared to it, a reduction of the checkpointing period is obtained for higher traffic loads (0.03 and 0.06) by using centralized markers (series CC). Thus, the scalability improvement at the protocol level

provides better scaling also with the traffic load, not only with the NoC size (as shown in the previous results).

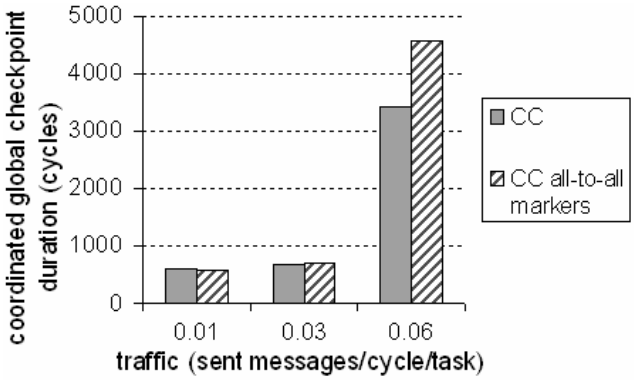
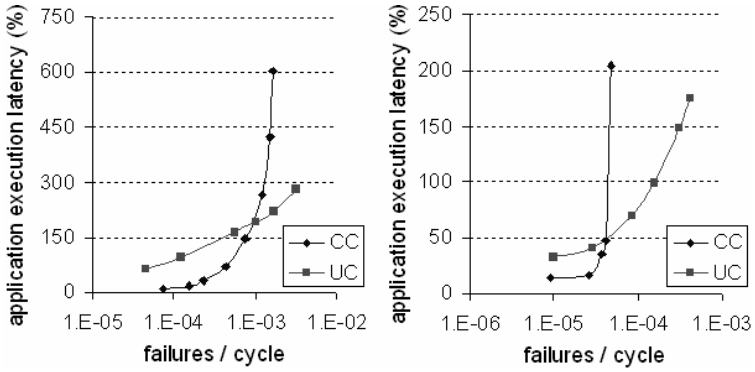


Fig. 4-28 Coordinated checkpointing duration for different traffic loads

A significant increase of the traffic load (related to the maximum throughput of the network) delays the synchronization messages in the NoC, increasing the checkpointing duration. Thus, the increase of the traffic increases the probability of late messages also by increasing the checkpointing duration. In the coordinated checkpointing approach, a new checkpointing phase begins only after the previous one has ended. Therefore, longer checkpointing durations determine the reduction of the number of checkpoints that can be performed in a given interval of time.

However, in order to be effective, a rollback recovery method must assure the existence of at least one recovery line between two successive failures. Therefore, the checkpoint frequency must be set according to the expected failure rate. In the next experiment the effectiveness of both methods is analyzed versus the expected failure rate.

A traffic of 0.01 sent messages/cycle/task is considered. Fig. 4-29.a depicts the application latency measured for the two rollback recovery approaches, for different failure rates.



a. traffic: 0.01 (sent messages/cycle/task) b. traffic: 0.06 (sent messages/cycle/task)

Fig. 4-29 CC vs. UC: Application latency

For lower failure rates, the coordinated checkpointing method has lower latency. Moreover, it is preferable in practice because of its simplicity and also to its smaller overhead. However, as the failure rate increases, the coordinated checkpointing approach becomes more and more ineffective. This happens when failures occur more often than the checkpointing duration. Therefore, a new checkpoint can not be established during the failure-free period. In these cases, rollbacks are always performed to the same old

recovery line and the execution of the application does not advance. The uncoordinated checkpointing becomes relevant for higher failure rates because of its ability to set incremental recovery lines from individual checkpoints. The cross point between the two curves occurs for 1E-3 failures/cycle.

The same experiment is performed using a six times higher traffic load, as shown in Fig. 4-29.b. In this case, the same trend is observed for the two recovery methods latency. Unlike in the previous comparison, the cross point between the UC and CC curves occurs for a more than one order of magnitude smaller failure rate. This can be explained by the increase of the coordinated checkpointing duration with the traffic load, as seen earlier.

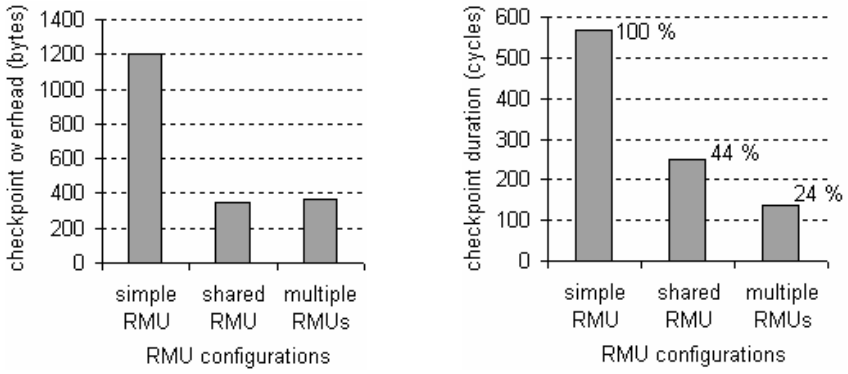
The simulations show that the coordinated checkpointing method is more efficient than the uncoordinated one for lower failure rates. It is also preferable because of its smaller overhead, especially for higher traffic loads. However, if the failure rate increases, the coordinated checkpointing becomes ineffective. In these cases, the uncoordinated checkpointing combined with message logging becomes relevant, despite its significantly higher overhead. Moreover, the effectiveness cross point of the two rollback recovery methods occurs much earlier as the traffic load increases.

4.6.2.4 Partition Configurations

This section presents the results obtained by simulation for the partition configuration presented in section 4.5.1, under different traffic loads, failure rates and NoC partition sizes. The results presented here use the coordinated checkpointing protocol presented in Fig. 4-14. Nevertheless, the partition configuration approach can also be applied to the uncoordinated checkpointing. The same coordinated checkpointing protocol is used for partitions in either the shared or multiple RMU configurations. The performance of the two recovery configurations is evaluated relative to the configuration with no partitioning, called *simple RMU*.

The first results are obtained by simulations on a 4x4 mesh NoC. The checkpoint overhead and duration are represented in Fig. 4-30. A uniform traffic load of 0.01 sent messages/cycle/task is used.

Significantly smaller checkpoint overhead and duration (Fig. 4-30) are obtained when the checkpointing is executed per individual partition (shared RMU and multiple RMUs). This occurs because the number of late messages decreases with the number of tasks participating to the global checkpointing, for the same traffic load. Shorter checkpointing durations also contribute in reducing the late message probability of occurrence.



a. Checkpoint overhead (bytes) b. Checkpoint duration (cycles)

Fig. 4-30 Checkpoint overhead and duration for different RMU configurations

The checkpoint overhead of the shared RMU configuration is comparable to the case when multiple RMUs are used. Thus, the shared RMU option can successfully replace (at a lower cost) the multiple RMUs when memory overhead is a limiting requirement.

The checkpoint duration influences the application overall latency and can impede the execution advancement when the failure occurrence period becomes comparable with the checkpoint duration. In Fig. 4-31 the execution latency of the tasks running on the same NoC-based system is represented as a function of the failure rate. The failure rate is expressed as failures/cycle/node.

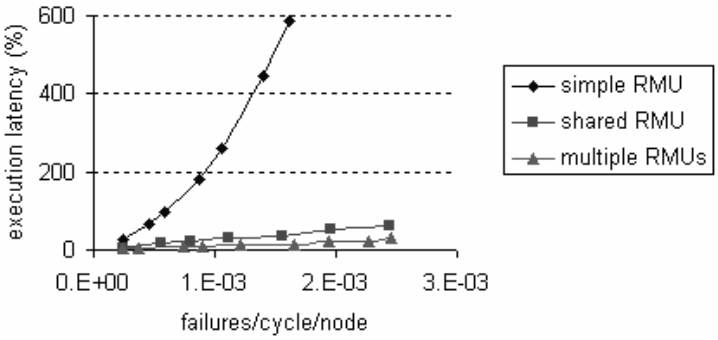
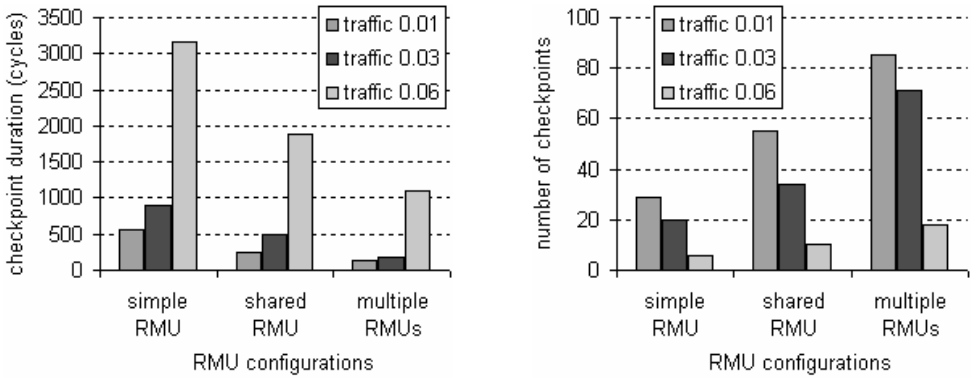


Fig. 4-31 Execution latency for different RMU configurations

As the failure rate increases, the simple RMU configuration becomes ineffective because of its high checkpointing duration. The other two optimized configurations maintain a lower latency for higher failure rates. Both the shared RMU and the multiple RMU configurations significantly enhance the system recovery capabilities for higher failure rates. In the considered case (4x4 mesh), the performance of the shared RMU case is comparable to the case of multiple RMUs, in spite of its lower cost.

With the following simulation we analyze the traffic load influence on the optimized RMU configurations. The checkpoint durations for the three RMU configurations are presented in Fig. 4-32.a, considering three traffic loads (0.01, 0.03, and 0.06 messages/cycle/task).

For the considered traffic loads, the most effective is the multiple RMU configuration, followed by the shared RMU one. Nevertheless, for higher traffic rates, the performance difference between multiple RMUs and shared RMU slightly increases. We also represented the number of checkpoints that can be taken in a given period of time, in Fig. 4-32.b. It can be observed that it decreases with the increase of the checkpointing duration.



a. Checkpoint duration (cycles) b. Number of checkpoints

Fig. 4-32 Checkpointing parameters for different traffic loads and RMU configurations

For very high traffic loads, the checkpointing duration increases significantly and the number of checkpoints decreases, therefore the overall execution latency increases faster when the failure rate increases. Consequently, the simple RMU configuration becomes ineffective for lower failure rates that in the case of lower traffic loads. Thus, the optimized RMU configurations are even more desirable for the applications having higher traffic loads.

Using the following simulations, we analyze the impact of the NoC size on the checkpointing duration for different RMU configurations. For doing this, we consider PEs partitions of the same size (2x2 PEs), but on a NoC size four times larger, an 8x8 NoC, as represented in Fig. 4-33.a.

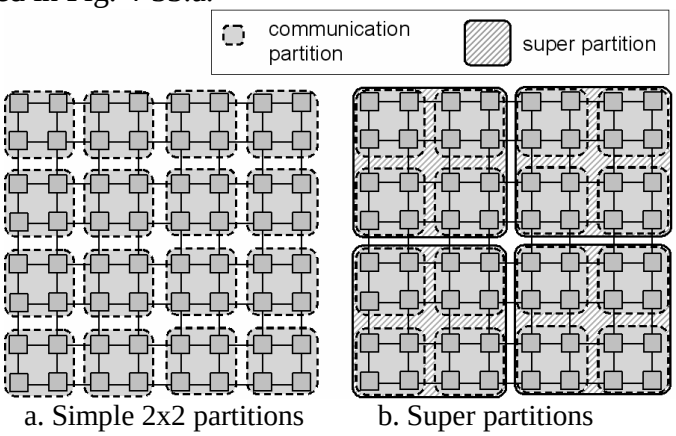


Fig. 4-33 8x8 mesh NoC with 2x2 partitions

Sixteen RMUs are necessary for the multiple RMU configuration. As a solution for reducing this high extra cost without a significant performance reduction, several shared RMUs can be used. The NoC partitions can be grouped in super partitions, as shown in Fig. 4-33.b, each handled by a shared RMU. Thus, only four RMUs instead of sixteen are necessary. The checkpointing durations obtained using the four recovery configurations are represented in Fig. 4-34.

We notice that the shared RMU configuration reduces the checkpointing duration to about 44% compared to the simple RMU configuration case. When compared with the 4x4 NoC case (with the same traffic load), we can appreciate that the shared RMU maintains its capability of reducing the checkpointing duration to more than a half. As a conclusion, the shared RMU approach scales at least as well as the simple RMU.

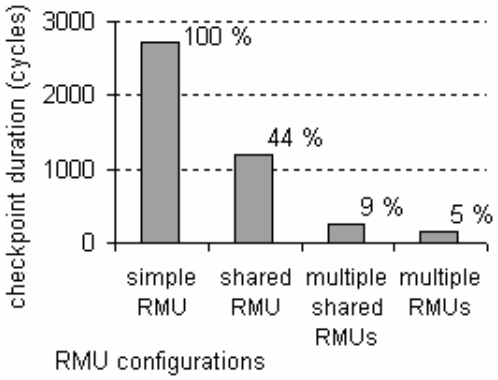


Fig. 4-34 Checkpointing duration for 8x8 NoC with different RMU configurations

When the multiple RMU strategy is employed, a more important reduction is observed (5%) when compared to the 4x4 NoC (24%, see Fig. 5). In fact, the performance

obtained by the multiple RMU configuration when compared to the simple RMU case strongly depends on the ratio between the partition size and the NoC size (1/4 for the 4x4 NoC and 1/16 for the 8x8 NoC). However, this important reduction is obtained at the cost of several RMUs (one for each partition).

As for the multiple shared RMU configuration, the performance is about 9% of the simple RMU configuration. The performance loss is about 4% with the costs reduced to a quarter, when compared to the multiple RMUs configuration.

Note that in practice it is not necessary that the communicating tasks are mapped on neighboring PEs or that the partitions are regular or have the same size. Moreover, different combinations of the proposed recovery configurations can be used for different NoC partitions, depending on the application specifics.

4.6.3 Limitations

There are two main limitations in the model we use, which are in fact due to the generality of the model: one concerns the application model and the other one the checkpoint moment.

The application model is limited because no synchronizations among tasks are considered. The model can be enhanced considering that a message can not be sent before the receipt of other messages, for instance. In our model, the moment of message sending is modeled, but there are no restrictions concerning the message receipt moment. The model can be restricted by imposing also the message receipt moment; thus, the task blocks until it receives the specified message, if the latter has not already arrived at the node. However, imposing more restrictions about the application restrains its generality. On the other hand, for each specific application, more restrictions increase the simulation accuracy.

The second limitation regards the moment when the checkpoint can be taken. In the current model, a checkpoint can be taken at any time, at pseudo-regular established moments in the case of uncoordinated checkpointing, and at RMU requests for the coordinated checkpointing. This assumption is realistic when a general checkpoint can be done, for example at OS level. It corresponds to the case when the application state content can be well delimited at any moment and its size is small. If this is not the case, the checkpoint must be done at the application level, which implies the contribution of the programmer. In this case, only well established points can be used to save the application state; outside these points, there is no guarantee that the state can be completely saved (for instance, a heap zone can be temporary allocated between two checkpoints) or, if it can, the state size can have great fluctuations from one checkpoint to another. In such a case, as the checkpoint can not be taken any time at once, the protocols must be adjusted (i.e. the early messages can not be avoided). A potential solution is presented in [BMPS03], where non-deterministic events that influence early messages are logged and replayed at sender. Adding restrictions about the checkpointing moments adds more variables to the exploration space. However, they must be present when special applications are targeted and the checkpoint can not be taken at the OS/middleware level.

4.7 Conclusions

This chapter presents the effectiveness vs. cost (in terms of overhead and latency) of checkpoint and rollback recovery protocols and their impact on the NoC performance.

Two main types of protocols were developed (uncoordinated and coordinated checkpointing) and scalability improvements were applied to them.

Concerning the coordinated checkpointing protocol, we developed a protocol allowing both blocking and non-blocking modes, which can be dynamically set, depending on the actual traffic load and expected failure rate. Also, the protocol allows performing the same global checkpoint with subsets of tasks blocking their normal execution and the others, not. This feature is very useful when heterogeneous tasks (in terms of QoS and data integrity requirements) share the NoC. This is a realistic case in NoC many core mobile and multimedia system, where certain tasks should not be blocked as they perform real time computation, while others may incur slight blocking especially for entertainement applications.

A second flavor of the coordinated checkpointing protocol was also developed, which improves its scalability by reducing the number of broadcasts. It maintains the same B-NB features as the first one.

A comparison between the uncoordinated and coordinated checkpointing protocols in terms of efficiency and cost was performed. Our simulations show that the coordinated checkpointing method is more efficient than the uncoordinated one under “normal” conditions of operation (i.e. low failure rates) and is preferable because of its smaller overhead, especially for high traffic loads. However, under hard conditions (i.e. very high failure rate), the coordinated checkpointing becomes ineffective. In these cases, the uncoordinated checkpointing combined with message logging becomes relevant, despite its significantly higher overhead. The effectiveness cross point of the two rollback recovery methods occurs much earlier as the traffic load increases.

We presented different configurations of the NoC recovery management units (RMUs) for improving the rollback recovery latency and overhead. The proposed recovery configurations exploit the localization of traffic in NoC systems. We analyzed these configurations under different traffic loads and failure rates and compared their effectiveness and costs. It results that the shared RMU approach is almost as efficient as the multiple RMU one, but more cost-effective.

A smart broadcast is proposed for use in the checkpoint and rollback recovery protocols. Its contribution in reducing the overall synchronization duration was estimated for the coordinated checkpointing. We presented results obtained by simulation on mesh NoCs of different sizes, which show a significant reduction of both the checkpoint latency and the recovery memory size overhead when using the smart broadcast.

Chapter 5. Reconfigurable Fault-Tolerant Inter-Layer Routing in 3D NoCs

This chapter presents a Reconfigurable Inter-Layer routing Mechanism (called RILM) for 3D NoCs. As exposed in the second chapter of this thesis, 3D NoCs with partial vertical connectivity are preffered, with each layer topology being different. These irregularities of 3D NoCs are taken into account by RILM, while much optimized 2D routing algorithms are used in each layer of the stack. We prove that RILM is fault-tolerant. In addition, if 2D routing algorithm in each layer is fault-tolerant, routing fault-tolerance is achieved for the entire layer stack.

First, we state on assumptions about: 2D and 3D topology, 2D routing policies in each layer and NoC element failures. Then, the 3D routing algorithm composed using RILM is presented. The next sections expose the initial configuration and the reconfiguration of RILM, both for the particular case of undirected-graph topologies and for the general case of mixed-graph topologies. RILM builds vertical node trees (VNTs) that are used for reconfigurations. Simulation results show the reconfiguration effectiveness. The impact on routing latency of topology and traffic patterns is also analyzed.

5.1 3D NoC Model and Assumptions

We consider a 3D NoC composed of several layers connected by vertical links that respects the following assumptions. Note that these assumptions held in the case of node or link failures that may occur. In the network routing context, by *nodes* we denote the routers.

A5-1. Between each two adjacent layers, there are at least two non-faulty vertical links (one for each direction) connecting non-faulty nodes at both ends, i.e. the stack of layers is not partitioned.

A5-2. All non-faulty nodes in the same layer are reachable from one another, by non-faulty links, i.e. the layer is not partitioned. The 2D routing algorithm of the layer can find such a path, if it exists; otherwise, the layer is considered as partitioned.

Assumptions *A5-1* and *A5-2* can be back-annotated with realistic data about layers and stack connectivity and with realistic failure probability that has to be computed for particular architectures.

A5-3. The routing of each 2D layer is deadlock-free [DS94] [CA95].

A5-4. In case of non-faulty links and nodes along the route, messages between any two non-faulty nodes are delivered to the destination in a predicted maximum time.

In case of failures, we assume that:

A5-5. Nodes and links are fail-silent [ALRL04]. A fail-silent component is a perfectly self-checking component that, as soon as any internal fault is activated, inhibits all outputs and ceases to provide any service.

A5-6. All non-faulty neighbors of a failed node are aware of the failure. "I'm alive" messages can be used to detected node failures. Additional detection and diagnosis steps are necessary to detect link failures [AN05].

A5-7. Both end nodes connected through a failed link are aware of the link failure.

A5-8. The routing algorithm in each 2D layer is fault-tolerant.

Observation: RILM can also be used with non-fault-tolerant 2D routings, while still tolerating vertical link failures and ensuring the reconfiguration of inter-layer routes. However, for achieving a fault-tolerant routing for the entire 3D stack, 2D fault-tolerant algorithms are necessary.

Note that no assumption is made about the topology of the 2D layers in the 3D stack. They can have any regular or irregular topology, as long as their routing algorithm complies with assumption *A5-8*. Besides, the fault-tolerant capability of the routing of each layer must be sufficient to the expected failure rate; as an extreme (an idealistic) example, if no failures can occur in one layer, the respective 2D routing could be non fault-tolerant at all, without affecting the overall fault-tolerance capability.

5.2 3D Routing with RILM

5.2.1 Routing Algorithm

The basic idea of the 3D routing algorithm using RILM is to move a message to the destination layer first (if not already there), then route it in this layer, to the destination node. To reach the destination layer, the message must span through other layers, called *intermediary* layers. To leave an intermediary layer for the next one, the message may need to move first inside the current layer in order to reach a vertical node that has an

output port to the next layer. For intra-layer routings, the routing algorithm of the current 2D layer is used.

The 3D global routing algorithm is presented in Algorithm 5-1. We denote the source and the destination of a message by *S* and *D*, respectively, and the current node by *C*. By *z* we denote the vertical dimension; the index of the layer a node *N* belongs to is denoted by z_N. We denote by V_{To}/V_{From} a vertical (or 3D) node that has output/input ports to/from vertical links. If the direction is specified, the vertical node is denoted by: V_{ToUp}, V_{ToDown}, V_{FromUp}, $V_{FromDown}$. For example, in Fig. 5-1, node 1 is V_{ToDown} and node 13 is both V_{FromUp} and $V_{FromDown}$. Node V_{To} in the 3D routing algorithm is determined in a specific way that is detailed in section 5.3.

Algorithm 5-1. 3D Routing – Global View

```
1: while z_D <> z_C
2:   if z_D upper of z_C
3:     then V_To = V_ToUp
4:     else V_To = V_ToDown
5:   end if
6:   2D routing to V_To
7:   Move to the next layer
8: end while
9: 2D routing to D
```

Let's see the routing algorithm executed by a router upon the receipt of a message header (Algorithm 5-2). The decision about the vertical node V_{To} to be used in each layer depends on the arrival node in the respective layer. This node becomes the message source in the layer, noted by S_{2D}. A supplementary field is added to those already used for the 2D routing implementation, namely D_{2D}, which designates the destination of the message in the current layer. D_{2D} changes in each layer at S_{2D} and is either a vertical node or the final destination (lines 1-11). Initially and just before moving to a new layer (lines 16-23), D_{2D} is set to null (line 17), indicating that the actual value has to be set. When the message reaches the final destination, it is delivered to the respective IP (lines 12-15).

Algorithm 5-2. 3D Routing – Algorithm Executed by a Router

```
1: if D_2D = null
 2:   if z_D = z_C
 3:     D_2D = D
 4:   else
 5:     if z_D < z_C
 6:       D_2D = V_ToUp
 7:     else
 8:       D_2D = V_ToDown
 9:     end if
10:   end if
11:   2D route to D_2D
12: else
13:   if D_2D = C
14:     if z_D = z_C
15:       Deliver to local IP
16:     else
17:       D_2D = null
18:       if z_D < z_C
19:         Route to up
20:       else
21:         Route to down
22:       end if
```

```
23:     end if
24:   else
25:     2D route to D2D
26:   end if
27: end if
```

Note that a 3D node may not necessarily have both a link towards upper layer and another one towards lower layer. Therefore, two vertical nodes must be known by each node in the 3D NoC, one for each vertical direction: V_{ToUp} and V_{ToDown}. A single vertical node is enough for the uppermost and lowest layers of the stack. The way V_{ToUp} and V_{ToDown} are determined is detailed in the following section.

5.2.2 3D Routing Example

In Fig. 5-1, a 3D routing example using the algorithm presented above is depicted. Suppose that the source of the message is node 34 in layer L_2 and the destination is node 12 in layer L_0. Let's assume some nodes and links are faulty. Faulty nodes are marked in dark (8, 24 and 31) and faulty links are marked with x-es.

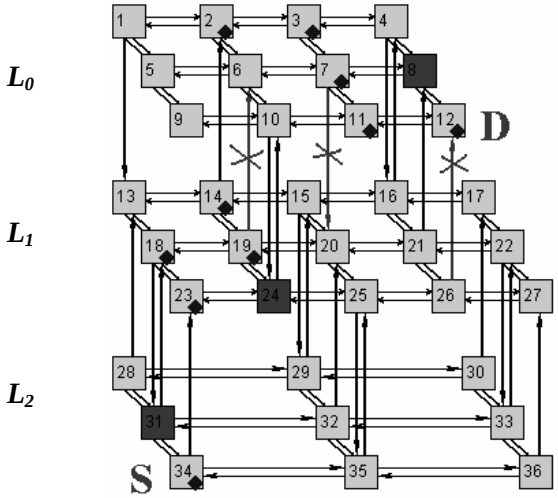


Fig. 5-1 3D NoC Routing Example

The route is determined according to the proposed routing algorithm. First, the routing mechanism must route the message towards the destination layer, L_0, i.e. upwards. At the beginning, D_{2D} is null, therefore the direction is decided for the first time at the source node, 34. Since $0 < 2$ ($z_D < z_C$), a V_{ToUp} node must be used. Node 34 is a node that belongs to this category. Normally, the message must be routed to the V_{ToUp} node, by using the 2D routing algorithm of the current layer, L_2. But in this case, the message is already in node 34. Anyway, the routing algorithm is executed again at node 34. This time, $D_{2D} = C$, that is to say the message has reached a vertical node for this vertical direction. As this is not the final destination, D_{2D} becomes null (for the next layer). Since the direction is up, the message will be routed up, to node 23, in layer L_1.

To move to layer L_0, the vertical node 14 (the closest one, in this case) is used; therefore D_{2D} is set to 14. To reach node 14, the 2D routing algorithm is used in layer L_1; thus the message passes through nodes 18 and 19. From node 14, the message moves in layer L_0 and reaches node 2. Before moving to layer L_1, D_{2D} is set to null. At node 2, as the message is already in the destination layer ($z_D = z_C$), the 2D destination of the message becomes the final destination: $D_{2D} = 12$. By using the 2D routing in layer L_0, node 12 is

reached, passing through nodes: 3, 7 and 11. Finally, at node 12, the message is delivered to the local IP.

To summarize this routing example, (S_{2D} and D_{2D}) in each layer are: (34, 34) in L_2, (23, 14) in L_1, and (2, 12) in L_0. The nodes the route passes by are marked with diamonds (Fig. 5-1): 34, 23, 18, 19, 14, 2, 3, 7, 11 and 12.

5.2.3 3D Routing Correctness

Let's prove the correctness of the 3D routing algorithm, for any topology, layer size and layer stack size, as long as the assumptions in the system model are met. Since the 3D routing algorithm is obtained by 2D routing algorithms composed using RILM, its properties strongly depend on the 2D routing algorithm properties.

Theorem 5-1. a) The 3D routing algorithm is scalable to any number of layers, since it is composable, by using 2D routing in each layer. b) The 3D routing algorithm is deadlock-free for one packet if all 2D routing algorithms used in layers are deadlock-free. c) The 3D routing algorithm is fault-tolerant if all 2D algorithms are fault-tolerant.

Proof. The message passes through a number of l layers, where $l = |z_S - z_D| + 1$. This is a consequence to the fact that the message is vertically routed only up or only down; there are no coming backs to the previously visited layers.

In each layer L_i, where $i = \overline{z_S, z_D}$, the 2D routing of L_i is used. Thus, a number of l horizontal routes $\{RH_i\}$ are determined. These horizontal routes are all completely distinct from one another, as they are located in different layers.

The route between layer L_i and the next layer (L_{i-1} or L_{i+1}, depending if $z_D < z_S$ or vice versa) consists in one non-faulty vertical link. This short vertical route, denoted by RV_i is deadlock-free and fault-free. The $l - 1$ vertical links are distinct; since there are no coming backs to the same layer, so it is impossible to use again a vertical link that is already used in the same route.

The complete *3D route* (*3DR*) followed by the message is obtained by chaining the $\{RH_i\}$ and the $\{RV_i\}$ routes. They are all completely distinct. If all $\{RH_i\}$ are also deadlock-free, *3DR* is deadlock-free. If all $\{RH_i\}$ are also fault-free, *3DR* is fault-free. (Q.E.D.)

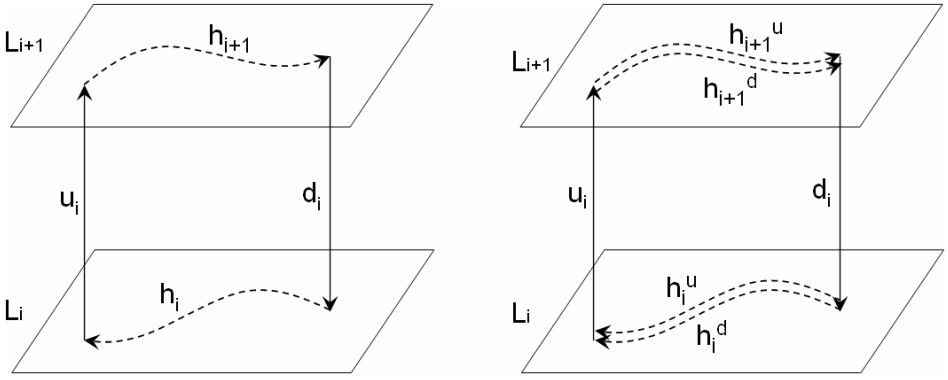
Theorem 5-1.b) states that a single packet (consisting of multiple flits) will not cause dealock either within a single layer or across multiple layers. The case of **deadlocks** of several packets (or messages, if formed of a single packet) is discussed in the following paragraphs.

The routing in 2D is ensured by the 2D routing algorithm of the layer, which is deadlock-free, according to assumption *A5-3*. Therefore, no deadlock occurs in 2D between intra-layer messages. Inter-layer messages behave like 2D messages in one particular layer. Besides, an inter-layer message passes through one layer only once, according to the 3D routing algorithm. Therefore, no supplementary channel dependencies can be created between an intra-layer and an inter-layer message. In consequence, deadlocks in such cases are treated by the 2D routing of the layer of the intra-layer message.

Regarding inter-layer messages that go in the same vertical direction, deadlocks in 2D layers are also treated by the 2D routing of each layer. New dependencies can be created at next layers, but this implies that the previous deadlock has already been solved. Thus, each deadlock is treated in its 2D layer. Inter-layer messages use also

common vertical links, but since these messages never return, cyclic vertical dependencies can not occur.

On the other hand, deadlocks can occur between inter-layer messages that go in opposite vertical directions, independently of the 2D routing policies in 2D layers. Such deadlocks are created by channel dependencies resulted from the routing of inter-layer messages that go up and those that go down. Such a dependency is depicted in Fig. 5-2.a. An inter-layer message M_u that goes to an upper layer can follow for example: route h_i, vertical link u_i, and route h_{i+1}. An inter-layer message M_d that goes to a lower layer can follow: route h_{i+1}, vertical link d_i, and route h_i (for the horizontal routes, only parts of them could be considered). If common parts of routes h_i and h_{i+1} are used by both messages (M_u and M_d), a deadlock occurs.



a. Cyclic channel dependency b. Avoid deadlocks by virtual networks in 2D

Fig. 5-2 Deadlock of inter-layer messages in opposite directions

In order to avoid such situations, two distinct virtual networks can be used in horizontal layers to break cycles: a virtual network for messages that go up and the other one for those that go down (Fig. 5-2.b). Horizontal routes in these virtual networks in layer L_i are denoted by h_i^u and h_i^d, respectively. Now, M_u will follow: route h_i^u, vertical link u_i, and route h_{i+1}^u. M_d will follow: route h_{i+1}^d, vertical link d_i, and route h_i^d.

In general, the routes of inter-layer messages that go up are formed only of $\{h_i^u\}$ and vertical links to up $\{u_i\}$. The routes of inter-layer messages that go down are formed only of $\{h_i^d\}$ and vertical links to down $\{d_i\}$, where $i = \overline{0, n-1}$ (n is the number of layers in the 3D stack). Thus, no common virtual channel is shared by inter-layer messages that go in different vertical directions, avoiding deadlock conditions.

Intra-layer messages share the virtual network with one of the inter-layer message types, either with those that go up or with those that go down, i.e. use $\{h_i^u\}$ only or $\{h_i^d\}$ only, where i is the index of the layer. This is possible since no deadlock can occur between intra-layer messages and inter-layer messages in one vertical direction. To balance the traffic between the two virtual networks in 2D layers, other options for intra-layer message routing should be explored.

5.3 Assigning Vertical Nodes for Routing with RILM

The 3D routing algorithm presented previously determines a 3D route composed of 2D routes, by using RILM. One of the most important decisions in each layer is to determine *the best* node V_{To} for inter-layer routing. The best vertical node is relative to many criteria (hop or geometric distance, congestions along the 2D route, congestions on the

respective vertical node etc.). A basic algorithm for selecting V_{To} vertical nodes (denoted by V_{ToUp} and V_{ToDown} in Algorithm 5-1 and Algorithm 5-2) is presented in this section, for 2D layers with any topology. The second part is dedicated to the adaptability feature of the algorithm, showing how the vertical node assignment can be optimized considering different criteria.

5.3.1 VNT Construction in 2D Layers

The main goal of the VNT (Vertical Node Tree) construction phase is to initially *assign* to each node in the layer two vertical nodes in the same layer (V_{ToUp} and V_{ToDown}), for inter-layer transfer of messages. The VNT mechanism can be used for assigning vertical nodes in any regular or irregular 2D topology. The vertical node assignment phase must be executed at the system initialization phase (but it may also be re-executed later in case of higher probability of failures developed during lifetime). The process of assigning vertical nodes is started at vertical nodes and spreads to the other nodes, using a controlled gossiping technique [PGC06]. Through this process, trees rooted in vertical nodes and called *vertical node trees* (VNTs) are built. The VNTs resulted are maintained in order to ensure the possibility of local reconfigurations (see subsection 5.3.2 and section 5.4).

Two strata of VNTs must be created in each layer of the 3D NoC, one for each vertical direction. For simplicity, a single direction is considered in the following paragraphs, i.e. a single stratum of VNTs.

Vertical links from one layer are likely to be fewer than the nodes in that layer; therefore, several nodes will use the same vertical link (we say nodes are *attached* to the respective vertical link). A VNT contains all nodes attached to the root node. The most economic implementation of a tree is to maintain at each node a *pointer to the parent* node.

The messages exchanged for VNT construction and reconfiguration are simply called *VNT messages*. For the VNT construction phase only a type of VNT messages is used, the *attach* messages, denoted by *mA*. They are also called *connect* or *join* messages. *mA* messages are sent only to neighbors of a node. The construction phase is initiated by vertical nodes, which set themselves as VNT roots and then send *mA* messages to their neighbors. Upon the receipt of an *mA* message, a node executes the Algorithm 5-3. Initially, all nodes are in the *nU* state, i.e. *unattached* or *unconnected* to any vertical node. After the VNT construction phase, all nodes must be in the *nA* state, i.e. *attached* or *connected* node in a VNT.

Algorithm 5-3. VNT construction

```
1: if node.state = nU // unattached
 2:   Connect(node, message)
 3:   node.state = nA
 4:   Send mA to node neighbors
5: end if
1: function Connect(node, message)
 2:   node.root = message.root
 3:   node.parent = message.source
4: end function
```

Upon the receipt of a *mA*, an unattached node will be attached to the root indicated in the message. It joins the same VNT as the message sender, which is one of the current node neighbors. The sender becomes the parent of the current node and the current node becomes the child of the sender in the same VNT. The new attached node, in turn, sends

*mA*s to its neighbors. The VNT construction phase is a finite process, as once attached to a VNT, nodes stop sending VNT messages.

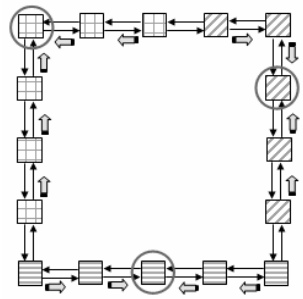
Observation: To assign a V_{To} node (*node.root* in Algorithm 5-3) to each node in the layer, the *node.parent* is not necessary. However, this field (*node.parent*) is necessary to maintain the VNTs, which are used for later reconfigurations.

5.3.1.1 VNT Examples

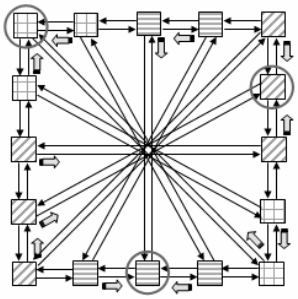
For different topologies, different types of VNTs are obtained, depending on the connectivity of the respective topology. The topology may have only bidirectional links between each pair of neighbors (*undirected graph*) or may also have only unidirectional links between certain neighbors (*mixed graph*). We will illustrate VNTs in both types of topology.

a) VNTs in 2D Undirected Graph Topologies (bidirectional links only)

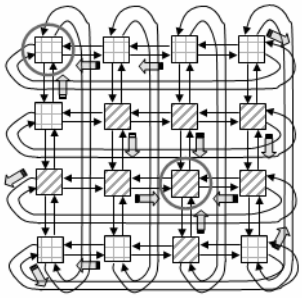
Examples of VNT strata for a single vertical direction are depicted in Fig. 5-3, for different undirected topologies. Vertical nodes are encircled and nodes in the same VNT are filled with the same texture. Block arrows point from each non-faulty 2D node to its respective VNT parent; these represent the VNT edges.



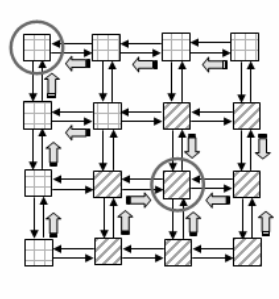
a. VNTs in ring topology



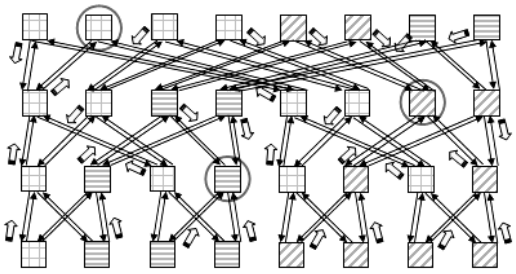
b. VNTs in spidergon topology



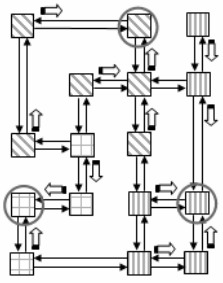
c. VNTs in torus topology



d. VNTs in mesh topology



e. VNTs in fat-tree topology



f. VNTs in irregular topology

Fig. 5-3 VNTs in undirected graph topologies

The first three examples (Fig. 5-3.a, b and c) present regular topologies with the same degree of all nodes: 2 for ring, 3 for spidergon and 4 for torus. Therefore, all VNT nodes will have at most the degree of the respective topology. The mesh is a particular case of torus, having the maximum degree of 4 (at the inside nodes) and smaller degrees for nodes on sides and corners.

Observation: The *degree* of a node in an undirected graph topology is the number of incident edges (bidirectional edges). In a mesh, for instance, the maximum degree of a node is 4. Nodes on sides have the degree of 3 and those in corners have the degree of 2. The degree of a

VNT node in undirected topologies is at most its degree in the topology. The number of VNT children can be up to the node degree (for the VNT root nodes). Non-root nodes can have a maximum of *node degree – 1* children (as one edge is used for the connection to the parent).

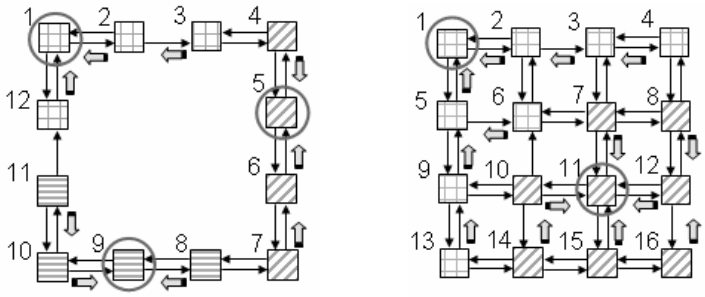
In Fig. 5-3.d, a fat-tree topology is considered. Because of the tree-like topology, VNTs rooted in nodes closer to the root level of the fat-tree spread faster than those rooted in lower levels. The example in Fig. 5-3.f presents an irregular topology, where nodes have different degrees.

Observation: Note that the VNT edges (block arrows in Fig. 5-3) do not determine the route of messages from 2D nodes to 3D nodes. They are used to maintain the VNTs. All messages (both intra- and inter-layer ones) from 2D to 3D nodes are routed using the routing algorithm of the respective 2D layer. However, for any regular or irregular undirected graph topology (as those depicted in Fig. 5-3), the VNT path from the root to any node in the VNT is bidirectional. In case of necessity, it could also be used for routing inter-layer messages to the VNT root. Thus, 2D link and node failures are tolerated by using the VNTs for 2D routing. However, this can be done only in intermediary layers, where the destination is the VNT root, but not in the destination layer, where the destination is independent of the VNT configuration.

b) VNTs in 2D Mixed Graph Topologies (bidirectional and unidirectional links)

VNTs are created also in mixed-graph topology, using the same algorithm presented above, as long as the assumption *A5-2* is met. Examples of VNT configurations in mixed-graph topologies are shown in Fig. 5-4. Only a ring and a mesh with incomplete set of links are depicted in this figure. In mixed-graph topologies, the distinction between VNT paths and the 2D routing paths is more obvious than in undirected graph topologies, as sometimes the VNT path from a node to its VNT root does not even exist (see the case of node 3 in Fig. 5-4.a).

Observation: In mixed-graph topologies, two different degrees are defined: *in-degree* and *out-degree*. The in-degree refers to the number of incoming edges to the node, while the out-degree refers to the number of outgoing edges from the node. The maximum number of VNT children in these topologies is limited only by the out-degree of the node (VNT root or not).



a. VNTs in ring topology b. VNTs in mesh topology

Fig. 5-4 VNTs in mixed graph topologies

Since there is a lack of bidirectional links, some nodes make attachments to vertical nodes that do not ensure an optimal path for routing inter-layer messages. For instance, node 3 in the mixed-ring topology (Fig. 5-4.a) is attached to vertical node 1. To reach it, inter-layer messages have to follow the clockwise direction from node 3 to node 1 (by nodes 4, 5, 6, 7, 8, 9, 10, 11 and 12). If the optimality of a path is measured in the number of hops, then this is obviously not an optimal path. Attaching to vertical node 5 would be more convenient for node 3.

Similar cases may also occur in other topologies, such as the attachment of node 4 in VNT_1 in Fig. 5-4.b. Joining VNT_{11} would be a better solution from the minimal distance

point of view. A method that allows more convenient reconnections of nodes in VNTs (for both undirected and mixed graph topologies) is presented in the next subsection.

5.3.1.2 VNT Construction Correctness

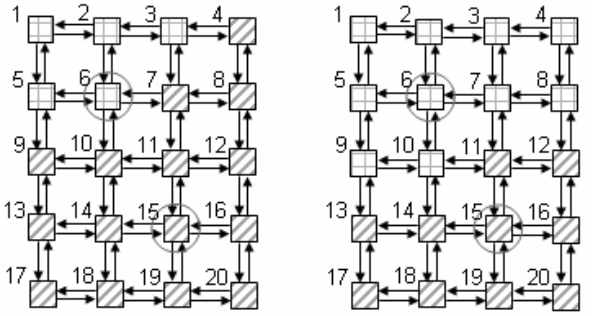
By executing Algorithm 5-3, all nodes in a non-partitioned layer will be attached to a vertical node. The following theorem is used to prove the VNT construction correctness.

Theorem 5-2. If vertical non-faulty nodes to one direction exist in one layer, the VNT stratum for the respective direction covers all nodes in the layer, for any topology of the layer, if the layer is not partitioned (assumption A5-2). That is to say, after the VNT construction phase, all nodes will be attached to a non-faulty vertical link for the respective direction.

Proof. Let V_{To} be any vertical node for the considered direction and let U_0 be any 2D (not vertical) node in the same layer. According to *A5-2*, at least one horizontal route between V_{To} and U_0 exists. We consider such a route from V_{To} to U_0 and refer it by *RH*. Let U_n be the first node in *RH*. U_n receives an *mA* from V_{To} when the VNT algorithm starts. If U_n is not attached, it attaches itself and spreads *mA* messages to its neighbors. Thus, an *mA* message reaches also to U_{n-1}, the next node in the *RH* route from V_{To} to U_0. On the other hand, if U_n is already attached, the attach message chain from V_{To} stops at U_n. This means that U_n has already spread *mA* messages to its neighbors when it has attached, so an *mA* have already reached U_{n-1}, too, as it is a neighbor of U_n. Thus, in both cases (U_n not attached or already attached), an *mA* is sent from U_n to U_{n-1}. Applying the same reasoning for the following nodes in the *RH* route from V_{To} and U_0, an *mA* has been sent from U_i to U_{i-1}, where $i = \overline{n,1}$. Thus an *mA* is sent from U_1 to U_0, so it can attach itself if not already attached. As U_0 represents any 2D node in the considered layer, we showed that any 2D node of the layer receives (at least) one *mA* message, so it can attach itself to a vertical node (Q.E.D.).

5.3.2 Optimization

In Fig. 5-5 .a, a VNT configuration in a 2D layer is depicted. Two vertical nodes to the same direction are present in the layer: node 6 and node 15.



a. Initial VNT configuration b. Balanced VNTs

Fig. 5-5 Different VNT balances

In this example, fifteen nodes are attached in VNT_{15}, while only five in VNT_6: nodes 1, 2, 3, 5 and 6. This may not be an optimal repartition of the 2D nodes to the 3D nodes in the layer, especially if the throughputs of the two vertical links are not proportional to the vertical traffic injected in them. However, this in real-life systems, several factors can contribute to uneven repartition of nodes in VNTs. For the example in Fig. 5-5 .a, the

uneven repartition can be due to an earlier start of VNT construction phase by node 15 relatively to node 6 or to congestions around node 6.

A weight function added in the VNT algorithm balances the VNTs by the selected criteria. The weight function is also a solution to the non-optimal attach regarding the routing path length, as presented for mixed-graph topologies in subsection 5.3.1.1.b. A *weight* is associated to each attachment of a node in a VNT, indicating the worthiness of the respective attachment by the selected criterion. The optimum attachment is that with the minimum or maximum weight.

The VNT attachment phase can be completed with such a weight function, as shown in Algorithm 5-4 (the lines added in the algorithm are highlighted). A supplementary weight field must be stored at each node.

Algorithm 5-4. VNT construction with weight function

```
1: if node.state = nU // unattached
 2:   node.weight = message.weight + 1
 3:   Connect(node, message)
 4:   node.state = nA
 5:   Send mA to node neighbors
6: else // attached
 7:   if (node.parent = message.source)
 8:     node.weight = message.weight + 1
 9:     Send mA to node neighbors
10:   else
11:     if(message.weight + 1 < node.weight)
12:       node.weight = message.weight + 1
13:       Connect(node, message)
14:       Send mA to node neighbors
15:     end if
16:   end if
17: end if

1: function Connect(node, message)
 2:   node.root = message.root
 3:   node.parent = message.source
4: end function
```

The connection itself is not modified, but it is conditioned by the relation between the actual and the potential weights of the node in the current and the new possible VNTs, respectively.

A simple example of a weight function is presented here – the number of hops from the VNT root to the current node. It can be computed directly at each node, upon the reception of attach messages, by simply incrementing the parent weight value by one. The weight of the root node is 0. We implemented this function in the simulator, so that in case of lack of uniform propagation of VNT messages, it will be used to balance the VNTs.

Fig. 5-5 .b presents the balanced VNTs resulted by applying this weight function, starting from the VNT configuration in Fig. 5-5 .a. When receiving the attach message from node 6, nodes 7 and 10, already connected in VNT_{15}, disconnect themselves from VNT_{15} and rejoin VNT_6. This is happening because their weight functions in VNT_6 are smaller then those in VNT_{15} (1 hop from node 6 versus 2 hops from node 15). Nodes 8 and 9 also leave VNT_{15} and join VNT_6 (weight is 2 in VNT_6 and 3 in VNT_{15}). Reconnections can also be done in the same VNT, but by another parent. Regarding the routing or the traffic balance on vertical nodes, a reconnection in the same VNT presents no interest. However, this can be useful for other purposes, such as faster disconnection

information spreading in the VNT, in the case of a weight function equal to the number of hops from the root (as the height of the VNT is minimized).

After each reconnection due to the weight function, the node sends *mA* messages to neighbors informing them about the reconnection. The children of the node update their weight function and later can decide of other eventual reconnections.

Other types of weight function can be implemented in the VNT construction algorithm, depending on the requirements of the system. Another example of weight function is the number of nodes in VNTs. However, for its computation, centralized information about the entire VNT is necessary.

In some cases, uneven distribution of nodes on vertical links can be desirable. It is the case of nodes that incur (or just transfer from other levels) unequal traffic loads on vertical links. Adaptability can be achieved by transferring lower traffic loads on links with smaller number of fault-free TSVs, for example. In general, an adapted VNT weight function obtained from the analysis of the actual vertical traffic load induced by the nodes in the layer can "balance" the VNTs.

5.4 Reconfiguration in the Presence of Failures

When failures occur, systems reconfigure in order to restore the connectivity of the communication network [AN05]. Not only 2D routes are affected by failures in a layer. In the mean time, in case of failures, the VNTs are no longer consistent. Re-creating the VNTs in the new topology configuration with Algorithm 5-3 is a possible solution to obtain consistent VNTs. However, the VNT reconfiguration presented in this section can be done only locally, in the region(s) affected by the failure. From the point of view of nodes affected by failure, the reconfiguration is done in two phases: *disconnection* (or *detaching*) from VNT, then *reconnection* (or *rejoining* or *reattaching*) in VNT.

5.4.1 Detaching from VNT

VNTs are composed of nodes and the paths used by *mA* messages during node attaching phase. In case of failure along these paths, the VNT structure is considered corrupted. Nodes in the entire (sub)VNT of the failed node or link (from a parent to a child node) are *affected by failure*, *AF*. The roots of these (sub)VNTs are neighbors of the failed node/link. The neighbors of a failed element are aware of the failure, as stated in assumptions *A5-6* and *A5-7*. Therefore, the detaching phase of affected nodes is initiated by the neighbors of the failed element, called *detaching phase initiators*, *DPI*. They start the detaching phase by disconnecting themselves from the VNT and sending *detach* messages to their first-order neighbors. *Detach* (or *disconnect*) messages represent the second type of VNT messages and are denoted by *mD*. Similarly to *mA* messages, *mD* messages are sent only to neighbors of the sender.

Upon the receipt of an *mD* message, an attached node executes Algorithm 5-5.

Algorithm 5-5. VNT detaching phase

```
1: if message.type = mD
 2:   if node.state = nA
 3:     if node.parent = message.source
 4:       node.state = nD
 5:       Send mD to node neighbors
 6:     end if
 7:   end if
8: end if
```

If the node receiving the *mD* message is attached and the sender of the *mD* is its parent for the respective direction, the node detaches: it gets into state *nD* (node disconnected) and sends *mD* messages to its neighbors.

Let's identify the *AF* and *DPI* nodes in all possible cases of failure:

1. *2D node failure*. All nodes in the failed node sub-VNT are *AF*s. The *DPI*s are the child nodes of the failed node.
2. *3D (vertical) node failure*. If the failed node is V_{To}, all the nodes in its VNT(s) are *AF*. The *DPI* nodes are the child nodes of the failed one. If the node is a V_{From}, then the VNT(s) rooted in its corresponding V_{To} node(s) in the upper and/or downer layer are *AF*. The *DPI* nodes are the V_{To} nodes. The adjacent vertical links in both directions of the failed 3D node are no longer used.
3. *2D link failure*. If this link used to be a VNT edge, the child node and its sub-VNT are *AF*. The *DPI* is the child node.
4. *Vertical link failure*. The nodes in the VNT rooted in the V_{To} end node of the link are *AF*. The *DPI* node is the V_{To} node.

In cases 1, 2 and 3, beside the VNT reconfiguration that we present, the 2D routing must also reconfigure, but this aspect is not treated in this work; it is specific to each 2D routing policy.

Observation: In cases 1, 2 and 3, the failed element belongs to a 2D layer, therefore it can be part of two VNTs, each belonging to one of the VNT strata created in each layer (marginal layers have only one VNT stratum). Both strata need to be considered for reconfiguration.

We showed previously which nodes must detach after a failure and pointed the nodes that initiate the detach phase (all of them are all neighbors of the failed element, so they are aware of the failure). We show that the exact (sub)VNT rooted in the initiator node (DPI) detaches, i.e., the *AF*s.

Theorem 5-3. In case of a failure, the whole (sub)VNT rooted in the detach initiator node is detached and no other nodes are detached because of mDs sent by the initiator, for any topology of the layer.

Proof. Let I_D be the detach initiator node. Let A_0 be a node in the (sub)VNT rooted in I_D, denoted by VNT_{ID} First we demonstrate that A_0 is reached by an *mD* message from its parent, so that it can then disconnect. Since A_0 belongs to VNT_{ID}, there is a path of nodes $\{A_i\}$, where $i = \overline{n,0}$, from I_D to A_0 (the VNT connection path). In the detach initiation step, an *mD* message reaches from I_D to A_n. This node (A_n) detaches, since I_D is its parent, then sends *mD*s to its neighbors. The same process repeats at each node A_i in the VNT path, so the node A_0 is reached by an *mD* message from its parent and disconnects.

The second part demonstrates that if E is a node that does not belong to the VNT_{ID} (external node), it does not detach in the detaching phase initiated by I_D. Indeed, even if E receives an *mD* message from a disconnected node, it does not disconnect unless the message is sent by its parent (see Algorithm 5-5). Since E does not belong to VNT_{ID}, neither does its parent. Thus, E does not disconnect. In fact, the disconnection process is stopped at neighbor nodes of the VNT_{ID}. With the two parts of the demonstration we showed that exactly the VNT_{ID} detaches (Q.E.D.).

5.4.2 Reattaching in VNT

The second phase of the reconfiguration is the reattach (reconnection) phase. It is initiated by attached nodes that receive *mD* messages. These nodes are called *reattaching phase initiators*, *RPI*. They are indirectly affected by the failure, i.e. they do not need to detach and reattach, but participate in the reconfiguration process. *RPI*s are neighbors of

the detached (sub)VNTs and border the expansion of the detaching phase. Instead, they are potential initiators of the reattaching phase of their neighbor detached nodes.

The reattaching phase can be done differently for undirected (graphs with only bidirectional links) and mixed graph (unidirectional links also exist) topologies. A simpler algorithm can be implemented for the undirected case. However, the one proposed for mixed-graphs is general and works for both types of topologies (undirected and mixed graphs).

5.4.2.1 Reattach in Undirected Graph Topology of 2D Layers

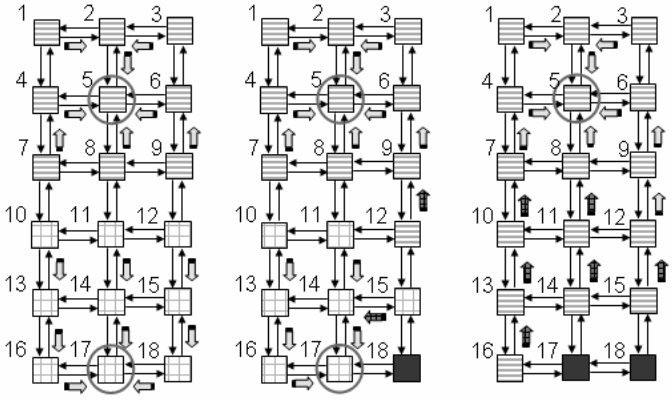
Upon the receipt of an *mD* message, a *RPI* sends *mA* messages to its neighbors. This is the initiation of the reconnection phase, as presented in Algorithm 5-6, lines 1-7.

Algorithm 5-6. VNT reattach in undirected graph topology

```
1: if message.type = mD
 2:   if node.state = nA
 3:     if node.parent <> message.source
 4:       Send mA to node neighbors
 5:     end if
 6:   end if
7: end if
8: if message.type = mA
 9:   if node.state = nD
10:     Connect(node, message)
11:     node.state = nA
12:     Send mA to node neighbors
13:   end if
14: end if
```

Upon the receipt of an *mA* message, a node in the *nD* state (disconnected node) will behave like the initially unconnected nodes (state *nU*) in the VNT construction phase (Algorithm 5-3). So, the algorithm must be completed to include this case (lines 8-14 in Algorithm 5-6).

In Fig. 5-6 an example of initial VNT configuration (for a single direction) and the configurations resulted after successive failures is presented for a regular mesh network.



a. Initial VNTs b. VNTs after node 18 fails c. VNTs after node 17 fails

Fig. 5-6 VNT construction and reconfiguration in regular mesh

Nodes from 1 to 9 belong to the VNT_5 rooted in node 5 and nodes 10 to 18 belong to the VNT_{17} rooted in node 17 (Fig. 5-6.a). Failed nodes are painted in dark. No nodes are failed in the initial configuration. If node 18 fails, its sub-tree nodes (15 and 12) detach. They rejoin VNTs through some of their non-faulty neighbors: node 15 rejoins VNT_{17} by

neighbor 14 and node 12 joins VNT_5 through neighbor 9. After an additional failure of 17, all nodes in the old VNT_{17} reconnect in VNT_5. The reconnection phase is initiated by border nodes of VNT_5 that are reached by *mD* messages from VNT_{17}. These nodes are: 7, 8 and 12. They spead *mA* messages to their neighbors and detached neighbor nodes rejoin VNT_5. These nodes are: 10, 11 and 15. After reattaching, they spread *mAs*, so the reconnection continues until the whole detached VNT is reconnected.

Using the following theorem, we prove the correctness of the reconfiguration phase in the case of undirected graph topologies.

Theorem 5-4. In the case of a failure in an undirected graph topology, all detached nodes from the VNTs they belonged to will reconnect, by joining another (or the same) VNT, using Algorithm 5-6.

Proof. Let B_0 be a detached node after a failure.

Case a. Node B_0 has at least one attached neighbor that is not affected by the failure, *E*. An *mD* message from node B_0 reaches node *E*. Node *E* will then send an *mA* message to its neighbors, according to Algorithm 5-6. The link between B_0 and *E* is bidirectional, hence an *mA* is sent equally to B_0. Thus, B_0 reconnects through *E*, which becomes its parent.

Case b. Node B_0 has no attached neighbor that is not affected by the failure. This means that B_0 is surrounded by disconnected nodes (and possibly, faulty nodes, but only non-faulty nodes are considered for the VNTs). Conforming to assumption *A5-1*, there still is a vertical node for the considered direction, V_{To}. As stated in assumption *A5-2*, a route between V_{To} and B_0 exists and, in the case of undirected graphs, it is bidirectional. Let the nodes on the path from B_0 to V_{To}, inclusive, be $\{B_i\}$, where $i = \overline{0, n}$, with $B_n = V_{To}$. Let B_j, where j is an integer in the interval $\overline{2, n}$, be the closest node to B_0, which belongs to the path to V_{To} and is not affected by the failure. In the worst case, B_j is V_{To}. As B_j is the closest node to B_0 not affected by the failure, B_{j-1} is affected, so B_j disconnects and sends *mD* messages to neighbors. According to case *a*, B_{j-1} can reconnect through B_j. After reconnection, is sends *mA* messages to neighbors, therefore B_{j-2} can also reconnect. The same operation repeats until the *mA* messages reach B_0, so it can also reconnect (Q.E.D.).

Case *a* can be considered as a particular case of *b* (for $j = 1$).

Observation. The nodes affected by failure can change their status several times during the reconfiguration process, depending on the actual traffic in the network and the structure of the VNTs. This incurs temporary steps until a stable VNT configuration is reached [Dij74].

5.4.2.2 VNT Reattach Phase in Mixed Graph Topologies of 2D Layers

Mixed-graph topologies present more general and complex reconfiguration cases. In these graphs, links from reattaching phase initiators (*RPI*s) to disconnected nodes may not always exist. Therefore, *mA* messages may not reach the detached nodes. In such cases, the initiation of the reconnection is done through a third type of VNT messages, denoted by *mR* (*reconnection*, *reattach* or *rejoin* messages). These messages are not sent to neighbors of the node, but are directed to the disconnected node; they are sent as ordinary messages. In undirected graph topologies, the whole VNT can be reached only by *mA* messages starting from any node that reconnects. For mixed-graph topologies, *mR* messages are used to direct the reconnection toward the root of the tree, as links towards child nodes can be unidirectional.

5.4.2.2.1 Algorithm

The reattach operations for mixed graph topologies are presented in Algorithm 5-7.

Algorithm 5-7. VNT reattach in mixed graph topology

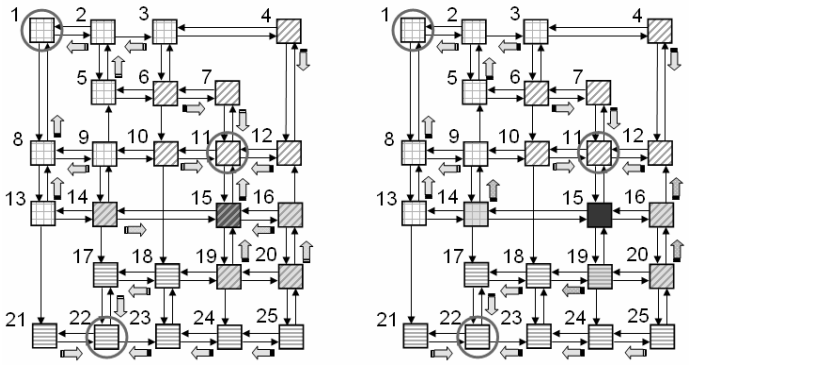
```
 1: if message.type = mD
 2:   if node.state = nA
 3:     if node.parent <> message.source
 4:       Send mR to source
 5:     end if
 6:   end if
 7: end if
 8: if message.type = mR
 9:   if node.state = nD
10:     Rejoin(node, message)
11:   else // node.state = nA
12:     message.root = node.root
13:   end if
14:   if message.destination <> node
15:     message.source = node
16:     Route message to message.destination
17:   end if
18: end if
19: if message.type = mA
20:   if node.state = nD
21:     Rejoin(node, message)
22:   end if
23: end if

 1: function Rejoin(node, message)
 2:   node.root = message.root
 3:   node.state = nA
 4:   Send mA to node neighbors
 5:   Send mR to node.parent
 6:   node.parent = message.source
 7: end function
```

Lines *1-7* refer to the reattach initiation phase. Lines *8-18* present the actions done at nodes on the route of an *mR* message. If the node is disconnected, it rejoins a VNT by the neighbor through which it received the *mR* message. If it is already attached, it just transforms the *mR* message for reconnection in its VNT. In any case, the new source of *mR* updates as the current node. Lines *19-23* present the *rejoin* when an *mA* message is received. Sending *mA* messages is still used in the algorithm, and they complement *mR* messages.

5.4.2.2.2 Examples

A scenario when *mA* messages are necessary is presented hereafter. Let's consider a node which has a single neighbor: its parent in the VNT. Consider also that the parent is not yet reconnected when receiving *mD* from its child node. Later, when the parent reconnects, if *mA* messages are not sent to neighbors, the child node remains disconnected.

An example for mixed-graph topology reconnection is presented in Fig. 5-7. The importance of routing *mR* messages like ordinary messages is shown. Even if the example considers an irregular topology, similar reconfiguration cases can occur in regular topologies that have links for only one direction between certain neighbor nodes.



a. Initial VNTs b. Reconfigured VNTs after node 15 fails

Fig. 5-7 VNTs in irregular mixed-graph topology

In this example, three initial VNTs are created: VNT_1, VNT_{11} and VNT_{22}, each rooted in a vertical node, which are encircled in Fig. 5-7. Nodes in the same VNT are filled with the same texture. Let's analyze the VNT reconfiguration in case node 15 fails. Neighbor nodes of node 15 are aware of the failure: nodes 11, 14, 16 and 19. Child nodes (14, 16 and 19) disconnect themselves from their parent, node 15. They will continue the spreading of *mD* messages and, as a result, node 20 also disconnects.

Let's see now what happens to the disconnected nodes. All of them will spread *mD* messages to their neighbors. We detail here a few possible reconnections initiated by the neighbors of node 14. Both direct and indirect reconnections are presented hereafter, in the initially designated VNT or not, as well as on-the-fly reconnections (reconnections of nodes through messages intended for the reconnection of other nodes).

- Node 14 spreads *mD* messages to its neighbors: 9, 13 and 17. Node 14 is a leaf node (it has no children), therefore its neighbors act like *RPI*s.
  - Nodes 9 and 13 will send an *mR* message to node 14, therefore node 14 can reconnect in VNT_1. Supposing these messages are directly sent to node 14 (1 hop), one of the *RPI*s (9 or 13, depending whose message reaches first to 14, for example) becomes the parent of 14.
  - Node 17 belonging to VNT_{22} will also send an *mR* message for reconnection in VNT_{22}. There is no direct link from 17 to 14. Let's say that this *mR* message is routed by the following nodes: 17, 18, 19, 20, 16, 12, 11, 10, 9 and 14. At node 18, which is attached, the message updates its source (to 18) and its VNT root (it remains 22 in this case). The *mR* message reaches then node 19.
    - Node 19, supposing it is not yet reconnected and using the *mR* coming from node 18, will be attached in VNT_{22}, by node 18, which becomes its parent. The same thing happens at the detached nodes 20 and 16. Node 20 reconnects by node 19.
      - Node 16 reconnects by node 20. When the *mR* message leaves node 16 for node 12, the source of *mR* is 16. As 12 is already connected (in VNT_{11}), it changes the *mR* message source to 12 and the VNT root to 11. Similarly, at nodes, 11 and 10, the source of the message is changed to 11 and 10; respectively. The VNT root remains 11. At node 9, the *mR* source changes to 9 and the VNT root to 1.
        - Node 14 can reconnect in VNT_1 upon the receipt of this *mR* message, even if it was initially sent for reconnection in VNT_{22}. Node 9 becomes the parent of 14, even if the *mR* message was originally

sent by node 17. The same reconnection of 14 is achieved by simply using the *mR* message sent by node 9.

An example of reconnection in the initial designated VNT by an *mR* message that can not be sent directly (1 hop) is the reconnection initiated at node 24 for node 19, if the *mR* message is routed through nodes 23 and 18.

At the end, all disconnected nodes reconnect by one of their neighbors (which will be the parent). However, the reconnection can be initiated by a different neighbor, as exemplified earlier. A possible reconfiguration after the failure is depicted in Fig. 5-7.b. Node 14 reconnects in VNT_1 by node 9, node 19 in VNT_{22} by node 18, and nodes 16 and 20 in VNT_{11}, by nodes 12 and 16, respectively.

5.4.2.2.3 Proof of Correctness

The correctness of the reattaching phase for the general case of mixed-graph topology is proved in the following theorem.

Theorem 5-5. After a failure, all detached nodes from VNTs of a layer with any topology reconnect, by joining another (or the same) VNT, using Algorithm 5-7.

Proof. It is sufficient to demonstrate the reattach of nodes in a single sub-tree affected by the failure, as all detached VNTs behave the same way.

Let's show first that once a node is reattached, the detached sub-tree reattaches. This is due to *mAs* that are sent immediately after attach (line *4* in function *Rejoin*), similar to the initial VNT creation. Any detached neighbor can reattach and inform its neighbors. Thus, the whole detached sub-tree reattaches.

It is sufficient now to show that there is at least an *mR* that reaches the root node, *R*, of the detached sub-tree.

R sends an *mD*. *mD* messages are sent to all neighbors, not only to children. Thus, *mD* reaches also the neighbor nodes of the detached area, D_1. These nodes are attached. Let E_0 be such a node.

Let F_m be the detached node that sent the *mD* to E_0 (see Fig. 5-8). Let F_j, where $j = \overline{1,m}$, be the path *mD* spread from *R* to F_m. If E_0 is attached, E_0 sends *mR* to F_m.

It is important to note that physically, there may not be a link from a node to its parent, but a link from the parent to the node is requested. Thus, *mA* and *mD* messages rapidly span the whole tree, going only one hop at a time. On the other hand, an *mR* message can not be always directly sent to a detached neighbor. This is why it is sent as a regular message.

If a direct link exists from E_0 to F_m, F_m attaches itself to E_0. Otherwise, a longer path from E_0 to F_m is followed, composed of nodes E_i, where $i = \overline{1,n}$. If E_1 is detached, it attaches itself first to E_0. E_1 sets itself as the source of the *mR* (see algorithm, lines *15-16*). Thus, E_2 can directly be attached to E_1 if it is detached. Consequently, all detached nodes along the path connect to the previous node in the path. When F_m is reached, it also connects to E_n. Then, F_m acts for its old parent (F_{m-1}) like E_0 acted for F_m, by sending an *mR* message to F_{m-1} (line *5* in function *Rejoin*). Then, an *mR* is sent from F_j to F_{j-1}, until j=0. Then, an *mR* is sent from F_0 to *R* (Q.E.D.).

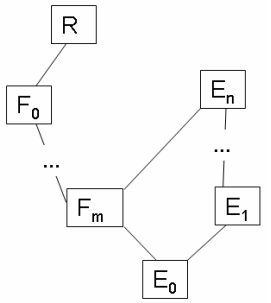


Fig. 5-8 VNTs. Reattach – *Theorem 5-5*

Note that E_i ($i = \overline{1,n}$) and F_j ($\overline{1,m}$) are not necessarily distinct. In such a case, a subtree reattaches earlier that its root.

In this paragraph we demonstrate that E_0 used in *Theorem 5-5* exists. Suppose there is no neighbor of the detached zone such as E_0. Therefore, all neighbors of D_1 are detached, which means that D_1 is just an island in a larger detached area, D_2. An E_0 neighbor of D_2 will determine nodes in D_2 to reattach. *mA* messages from D_2 must also reach D_1 (*A5-2*), otherwise D_1 is isolated. If D_2 is the largest detached area that includes D_1, E_0 exists (otherwise all nodes in the layer are detached, which means that no vertical links are available anymore; this contradicts assumption *A5-1*). Thus, E_0 exists.

Node Rejoining

Using the fail-stop (or fail-silent) model makes that a node, which self-detects errors, stops processing and consequently, the number of holes in the NoC (i.e. inactive nodes) should increase inexorably. Therefore, a mechanism to reactivate nodes which encountered a transient error and stopped should be considered. RILM can be directly used for this purpose. After a node failure, the node will be detached from the VNTs and excluded from routing. Then, if the test and diagnosis indicate that the failure was caused by a transient error, the node can rejoin the network. With RILM, the rejoining is simply started by the respective node, by sending detach messages to its neighbors (exactly like in the case of nodes affected by failures). Then, the reattaching phase will be initiated by its neighbors already attached. If none of them is currently attached, the reactivated node will be reattached later, when one of its neighbors will be attached. Regarding the routing algorithm of the concerned layer(s), they must be able to deal with node rejoining.

5.5 Properties and Evaluations of RILM in 3D NoCs

In the first part of this subsection we assess the complexity and overhead of the 3D routing relatively to the existing 2D routing, in terms of supplementary fields stored at each node and in each message and in terms of supplentary virtual networks to avoid deadlocks.

The reconfiguration capability of RILM resides in the VNTs. The overhead and complexity of construction and reconfiguration of VNTs (presented in the previous two sections) are also assessed, in terms of number of exchanged messages and their routing overhead.

- *3D Routing with RILM: Generality and Heterogeneity*
  Any type of routing algorithm used in the 2D layers can be integrated in the 3D routing by RILM. Routing algorithms can be different in different layers. If fault-

tolerant capabilities are not necessary in layers, simple routing algorithms can be used (such as XY in mesh topologies). However, for the layers where faults are likely to produce, fault-tolerant routing algorithms are required. Minimal fault-tolerant algorithms can be used for regular topologies, but if no such routing algorithm is available for the targeted topology, general algorithms using routing tables and dealing with irregular topologies can always be used [AN05].
For the 3D routing algorithm using RILM (section 5.2), nodes of the NoC do not need to keep global information about the topology (such as routing tables or lists of failed nodes or links), minimizing thus the routing memory overhead. Yet, if a 2D routing algorithm in a layer uses such approaches, it may remain as it is.

- *RILM Effectiveness*
  The 3D routing algorithm works for any number of layers and for very high number of missing vertical links (from design or because of failures). If the number of layers is *L*, then the minimum number of available unidirectional vertical links is:
  $VL_m = 2*(L-1)$ (5-1)
- *RILM Constant Overhead => Scalability*
  For the 3D routing, only a small and constant amount of data needs to be added in application messages and nodes. Thus, the routing algorithm scales to any number of nodes in each layer and any number of layers for a 3D application.
  - *Additional Fields in Messages or Packets*
    Each application message has a supplementary field containing the local destination, D_{2D} (i.e. V_{To}), used by the 2D routing in intermediary layers.
  - *Memory for VNTs*
    For VNTs, *two field sets* (one for up and the other for down) of (*state*, *parent*, V_{To}) must be kept at each node to define the node connection in a VNT. A weight field can also be added, if necessary.
  - *Virtual Networks to Avoid Deadlocks of Inter-Layer Messages*
    Separate virtual networks have to be used for inter-layer messages that go in opposite vertical directions, as presented in subsection 5.2.3. Thus, the overhead induced by a supplementary virtual network in each 2D layer must be considered for implementation. In fact, if the 2D routing in a layer already makes use of virtual networks, these have to be doubled in that layer: a set for the messages that go up and the other for those that go down. The overhead induced by these virtual networks in 2D may be considerable, but it does not depend on the stack size, maintaining thus the scalability feature of the routing algorithm. Also, no overhead must be added for the vertical links. There will be also a latency induced by the virtual network selection, but in the mean time, the latency experienced by messages when routed horizontally (inter-layer and local intra-layer messages) will diminish, because several resources will be available.
- *Messages for VNTs*
  The VNT messages are very short, containing two field sets (one for VNT to up and the other for VNT to down) of (*type*, V_{To}). The latency the VNT mechanism incurs is similar to that of the reconfiguration of the 2D fault-tolerant routing after a failure.
  The number of VNT messages exchanged for VNT construction and reconfiguration can be estimated as follows.
  - *Messages for Assigning Phase*
    The number of sufficient *mA* messages exchanged to achieve an initial attachment of all nodes in the layer is N_{mA}:

$$N_{mA} = \sum_{i=1}^{n} o_i \quad (5\text{-}2)$$

where n represents the number of nodes in the layer and o_i is the number of 2D output ports of node i. N_{mA} is independent of the number of output vertical links available for the layer.

For example, for a regular mesh, $N_{mA} \sim 4*n$, as each node has 4 neighbors (excepting corner and side nodes). If a weight function is used, additional *mA*s are exchanged to achieve a more convenient VNT configuration than the initial one.

- *Messages for Detaching Phase after Failure*

  In case of failure, the number *mD* messages exchanged is:

$$N_{mD} = \sum_{i=1}^{n_{AF}} o_i \quad (5\text{-}3)$$

  where n_{AF} is the number of nodes in the sub-tree affected by the failure (*AF*) and o_i is the number of output 2D ports of node i.

- *Messages for Reattaching in Undirected Graphs*

  The reconnection in undirected graph topologies (Algorithm 5-6) is done through *mA* messages. In this case, *RPIs* send *mA* messages to their neighbors. *AFs* also send *mA* messages after their reconnection. Thus, the number of *mA* messages used for reconnection in undirected graph topologies is:

$$N_{mA\_U} = \sum_{i=1}^{n_{NIR}} o_i \quad (5\text{-}4)$$

  where n_{NIR} is the total number of *nodes involved in the reconfiguration*, *NIR, and:*

$$NIR = RPI + AF \quad (5\text{-}5)$$

  All messages considered above are short messages (one hop).

- *Messages for Reattaching in Mixed Graphs*

  In the case of mixed-graph topologies (Algorithm 5-7), the reconnection is done through both *mA* (one hop) and *mR* messages. The latter span variable path lengths. The number of *mA* messages (sent after node reattachments) is:

$$N_{mA\_M} = \sum_{i=1}^{n_{AF}} o_i \quad (5\text{-}6)$$

  It can be observed that $N_{mA\_M} = N_{mD}$, as only reattached nodes send *mA* messages. The number of *mR* messages exchanged during reconfiguration in mixed-graph topologies is:

$$N_{mR\_M} = nn + (nAF - nCF) \quad (5\text{-}7)$$

  where *nn* is the number of (unidirectional) links from *AF* to *RPI* nodes (as each *AF* sends *mD* messages to neighbors and *RPI* respond by *mR*s). The second term (*nAF – nCF*) represents the number of *mR* messages sent to the old parent after the reconnection of each affected node. This number is equal to the number of affected nodes (*nAF*) minus the number of children of failed nodes (*nCF*).

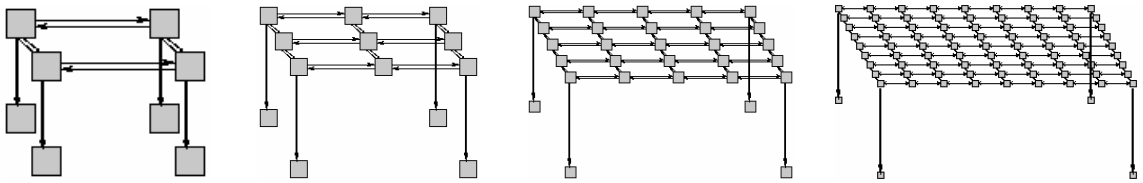
  The number of *mR* messages (N_{mR_M}) can be reduced. For example, suppose a node reconnects through its old parent. In this case, the parent is already connected. Therefore, sending the *mR* message from the node to the parent is not necessary and can be skipped.

Observation: The numbers of messages exchanged for initial attachments or reconfiguration estimated with the above formulas represents minimal numbers. In reality, these numbers can be greater, because of temporary attachments due either to weight functions or to the distributed nature of the algorithms (i.e. temporary states can be reached before stabilization).

5.6 Experimental Results for 3D NoCs with Mesh Layers

5.6.1 Simulation Environment

- *3D Topology*
  For simplicity, the 3D NoC architectures used for simulation are based on **generic 3D meshes**, noted by *LxRxC*, where *L* means the number of horizontal layers, each layer having *R* rows x *C* columns of nodes. In our simulations, different topologies are considered, with *L* up to 6, and *R* and *C* up to 9.
- *2D Links*
  Mixed graphs (both unidirectional and bidirectional links) and directed graphs (unidirectional links only) usually induce unequal utilization of links. Weight functions specific to each concrete topology must be implemented in these cases. Therefore, for the analysis generality, and since they are the most used in practice, undirected mesh graphs are considered for 2D layers; i.e. 2D links are **bidirectional**. The weight function given as example is section 5.3.2 is used to balance the VNTs in 2D layers.
- *Vertical Link Number*
  The most complete configuration of a generic 3D mesh (*LxRxC*) has VL_{adj} vertical **unidirectional** links between each pair of adjacent layers:
  $VL_{Madj} = 2*R*C$ (5-8)
  Thus, applying (5-8) for each of the (*L*-1) pairs of adjacent layers, the maximum number of available vertical links in the 3D NoC is:
  $VL_M = 2*R*C*(L-1)$ (5-9)
  The minimum number of available vertical links is the same as in the general case, given in formula (5-1): $VL_m = 2*(L-1)$.
  A minimum number of vertical links (VL_m) are always necessarily non-faulty, in order to ensure the connectivity between layers (assumption *A5-1*): at least two links are necessary between each pair of adjacent layers. The maximum number of failed/missing vertical links allowed is ($VL_M - VL_m$).
- *Vertical Link / Node Pattern*
  Vertical nodes to the same direction (to up/down) can be located in various patterns. We consider **regular patterns** delimited by **different number of hops** between vertical nodes, as depicted in Fig. 5-9 for vertical nodes/links to down.



a. 1 hop b. 2 hops c. 4 hops d. 8 hops

Fig. 5-9 Regular patterns of vertical link distribution

These patterns can be different for different layers. Even for the same layer, the pattern of the links to up can differ from that of the links to down. Therefore, different vertical link *distributions* can be obtained.

- *Simulator*
  The **simulator** presented in section 1.7.1 is used for RILM simulations. The **2D fault-tolerant routing** algorithm presented in subsection 5.6.2 is embedded in each node,

since the routing decisions can dynamically change. **RILM** is also embedded in each node. Using these two together, a 3D fault-tolerant routing algorithm is obtained.

- *Traffic Injection*
  Uniform traffic loads are injected in the NoC at each cycle and at each node, with the same probability, *p*. All messages injected have the same number of flits, *f*. The uniform traffic injected is denoted by: ***p* messages/cycle/node x *f* flits**, or simply ***pxf***. The destination of each message is usually randomly chosen, but we also consider some restrictions to obtain different rates of intra- and inter-layer messages.

5.6.2 2D Fault-Tolerant Routing Algorithm

The 2D fault-tolerant routing algorithm used in our simulations for 2D layers is presented in Fig. 5-10. It is designed for 2D mesh topologies and does not require routing tables. The routing algorithm is embedded in each node and the route is determined on the fly, using information about the faulty nodes. The algorithm uses a combination of the North-Last and South-Last routing strategies. Both strategies avoid deadlocks by numbering the links. The numbering ensures that there are no deadlocks if the message follows links with numbers that strictly increase. The routing algorithm uses two virtual networks, one for North-Last and the other for South-Last routing. Depending on the location of the destination relative to the source, one of the two virtual networks is selected. Fig. 5-10 shows a 2D routing example using this algorithm.

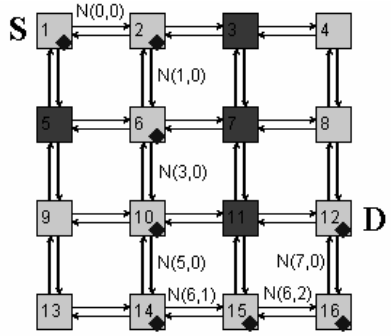


Fig. 5-10 2D fault-tolerant routing for meshes

Let's suppose four failed nodes: 3, 5, 7 and 11. The source and the destination of the message are nodes 1 and 12, respectively. As the destination is south of the source, the North-Last virtual network is used and the message is routed through the following links: N(0,0), N(1,0), N(3,0), N(5,0), N(6,1), N(6,2) and N(7,0).

An improved implementation of this algorithm for systems with many cores is presented in [CAZN10], where it is shown that no packets are lost for high percents of failures.

5.6.3 Simulation Results

Simulation results regarding two main aspects are presented in this subsection:

- VNT reconfiguration after failures, in terms of: number of nodes involved, duration and number of messages exchanged;
- Effectiveness of RILM 3D routing algorithm in a 3D NoC, considering different parameters and assessing their impact on the system performance.

5.6.3.1 VNT Reconfiguration after Failure

a) VNT Reconfiguration Expansion

Different vertical link patterns and different failure rates are considered in a 2D layer to analyze the VNT reconfiguration expansion, in terms of nodes involved.

For this analysis, the four patterns depicted in Fig. 5-9 are applied in a 9x9 mesh layer. For these patterns, the number of nodes involved in reconfiguration in the 9x9 layer was counted for several node fault multiplicities (from 1 to 5). The results are presented in Fig. 5-11. Each bar represents the number of the *Nodes Involved in Reconfiguration*, *NIR*. This number represents the sum of the number of *nodes Affected by Failure* (*AF*) and the number of *Reattaching Phase Initiators* (*RPI*).

Based on the results shown in Fig. 5-11, for different vertical link patterns and fault multiplicity, the following observations can be made. For a given fault multiplicity, *NIR* increases with the increase of the VNT hop distance. Indeed, for greater VNT hop distances, the VNTs are larger in terms of number of nodes contained, so the (sub)VNTs affected by the failure also contain a greater number of nodes. For a given VNT hop pattern, the *NIR* increases with the fault multiplicity, since several (sub)VNTs (eventually partially overlapped) are affected by the failure.

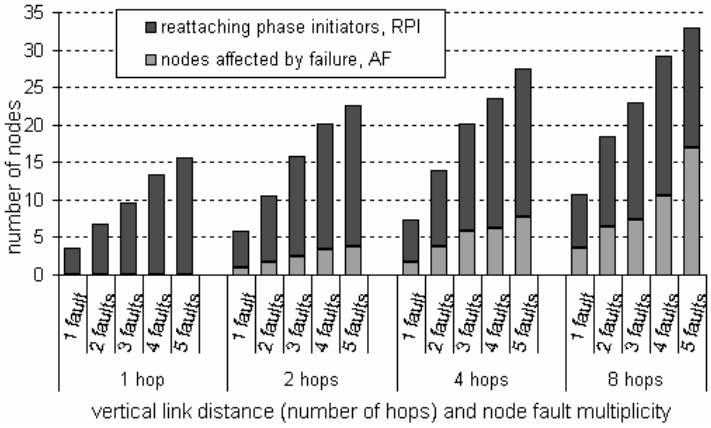


Fig. 5-11 Number of nodes involved in reconfiguration

For the *1 hop* pattern, no nodes need to reattach, since all nodes in the layer are 3D nodes. For the other patterns, *AF* increases with the fault multiplicity. It can be observed that the ratio *AF*:*NIR* increases with the increase of the VNT hop distance. This is explained by the fact that with the increase if the VNT size, the number of nodes in the VNT increases more when compared to the number of the VNT vicinity nodes (*IAF* nodes).

The results presented in Fig. 5-11 are averages computed on several simulations (a few dozens are enough to converge, for this topology size), where the failed nodes are randomly picked among the 81 nodes in the layer. For each particular fault case, the number of concerned nodes varies depending on the fault pattern, the affected VNT sizes and the topology.

For the same layer and patterns of vertical links, we analyze the impact of single failures of 2D and 3D nodes on the number of nodes involved in reconfiguration. The corresponding numbers of nodes are depicted in Fig. 5-12, series *2D* and *3D*. The average number of nodes involved in reconfiguration, independently of the failed node type (2D or 3D), is the third series of data, denoted by *all*.

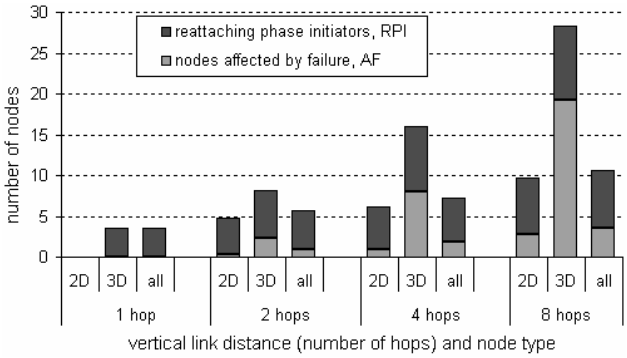


Fig. 5-12 Number of nodes involved in reconfiguration for single failures of 2D and 3D nodes

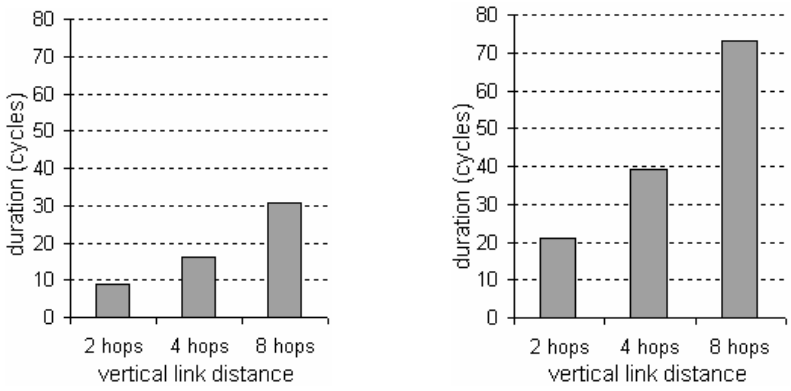
In the case of 3D node failures, nodes in adjacent layers can also be involved in reconfiguration, in the case of V_{From} nodes, as detailed in section 5.4.1. Their number varies depending on VNT configuration in the respective layers. For simplicity, only nodes in the same layer as the failed node are considered in this experiment, since only V_{To} nodes are taken into account.

The number of nodes involved in reconfiguration is greater in the case of 3D nodes, than in the case of 2D nodes. This is explained by the fact that the entire VNT is affected by 3D node failure, not only sub-VNTs, such as in the case of 2D node failures. Also, the probability of temporary attachments in the same VNT, which in the final configuration may not exist, is also higher.

As the number of hops in the vertical link pattern increases (Fig. 5-12), the gap between the *2D* and *3D* series becomes significant. However, the average number of nodes involved in the reconfiguration (*all* series) is closer to the 2D cases. Large VNTs are determined by relatively small number of 3D nodes. Therefore, if all nodes in the layer fail with the same probability, the probability that a 3D node fails is significantly smaller than the probability that a 2D node fails. For example, for the *8 hops* pattern (Fig. 5-9.d) applied in the 9x9 layer, the 3D node failure probability is only 5% (4 nodes out of 81) and the 2D node failure probability is 95% (77 nodes out of 81). This explains why the number of involved nodes in reconfiguration remains relatively low.

b) VNT Reconfiguration Duration

The same four patterns of vertical links (Fig. 5-9) applied in the 9x9 layer are used to assess the duration (in cycles) of the VNT reconfiguration after a single failure. The results are presented in Fig. 5-13.a.



a. VNT reconfiguration duration b. VNT construction duration

Fig. 5-13 VNT reconfiguration vs. VNT construction durations

As expected, the duration of the VNT reconfiguration increases with the distance between the vertical links. However, this duration is independent on the layer size. For comparison, the duration of the initial VNT construction (5.3.1) in the 9x9 layer is given in Fig. 5-13.b. Both durations are measured with no other traffic in the NoC beside the VNT messages, as the durations can vary with the existing traffic. The reconfiguration duration is smaller (with about 50%) than the VNT construction duration. In case of runtime reconfiguration of the system after failures, the VNT reconfiguration is preferable to the VNT reconstruction. The latter would also comprise the phase of informing all nodes to pass in the unattached state before starting the VNT construction, phase that is not considered in these simulation measurements.

The VNT runtime reconfiguration affects only inter-layer messages. If a 2D node or link fails, the 2D destination (V_{To}) of an inter-layer message is not affected by the failure. Even if its source will have another V_{To} after the reconfiguration, the initial one can be used to move to the next layer. Note that the VNT edges are not used for routing; the 2D fault-tolerant routing is used for this purpose. If a vertical node of link fails, inter-layer messages already headed toward that local destination (the vertical node) are lost (or, if the local destination is changed along the route, deadlocks can occur). The smaller the VNT reconfiguration duration is, the lowest the probability of headed inter-messages towards a failed V_{To}. Therefore, a minimum VNT reconfiguration is desirable. However, the VNT reconfiguration is only a part of the system reconfiguration. Messages can also be lost because of the failure of their final destination, the task migration from the failed node, the 2D routing reconfiguration etc.

c) Number of Exchanged Messages during VNT Reconfiguration

Single Failure

The reconfiguration overhead to the traffic load in the NoC depends on the number of VNT messages. The same four patterns of vertical links (Fig. 5-9) are considered for comparison. The numbers of messages exchanged during reconfiguration are given in Table 5-1. For the initial VNT construction in the 9x9 layer, 288 VNT messages are exchanged. Thus, the overhead in terms of number of exchanged messages is also significantly smaller in the case of a reconfiguration.

Table 5-1 VNT messages for reconfiguration

Vertical link pattern	VNT messages
1 hop	***0***
2 hops	***2.04***
4 hops	***7.17***
8 hops	***20.06***

Considering the 3D vs. the 2D node failures, as in the case of the number of nodes involved in reconfiguration, the number of exchanged messages is higher, but this compensates with the smaller probability of occurrence of 3D node failures (considering their number when compared to the number of the 2D nodes).

Multiple Failures

With the following simulations, the case of multiple node failures in one layer is analyzed (Fig. 5-14). The same 9x9 mesh layer is used for simulations. The pattern of vertical links is *2 hops*, as in Fig. 5-14.b.

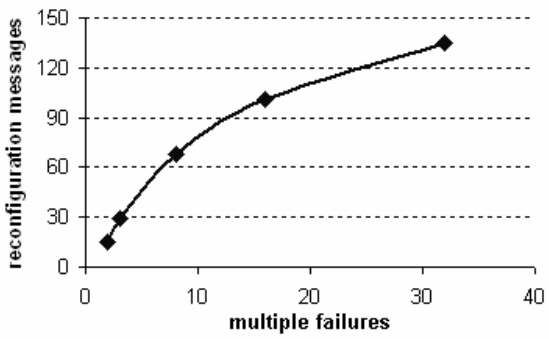


Fig. 5-14 Number of reconfiguration message vs. multiple failures

As expected, the number of exchanged messages for VNT reconfiguration increases with the number of failures. However, for lower multiplicities of the failure rate, the increase is more significant than for very high multiplicities. In fact, for lower multiplicities, the probability that the failed nodes are close to each other is low. Thus, the reconfiguration is done as for single failures. On the other hand, for high number of failures, since the probability that the nodes are close is higher, reconfigurations overlap (the same message can be used for different reconfigurations). Besides, for very high failure multiplicity, the number of nodes that need to reconnect is smaller.

5.6.3.2 Evaluation of 3D Routing

Using the following simulations, the performance of the 3D routing is analyzed, in terms of message latency, considering different variables such as number and pattern of available vertical links, traffic load and node failure rate and pattern.

The *latency* of a message is the time to propagate the first flit (to the destination node), considered from the moment the message is ready to be injected in the NoC. The latency also includes the time of waiting for the effective injection, if the NoC resources at the entry point are not available right away.

a) Impact of Available Vertical Links

Depending on several factors, such as: technology for physical implementation, design complexity, and number of failures that occur, different rates of vertical links are available in the system. The rate of vertical links is considered with respect to the maximum number of vertical links from the architectural point of view.

We consider a 3x5x5 3D topology: 3 stacked layers, each having a 5x5 mesh topology. In the 3x5x5 topology, the maximum number of unidirectional vertical links is 100: links in each vertical direction connect each node with its homologous node in adjacent layers. With smaller rates of vertical links (considering the maximum possible), the maximum throughput of the NoC decreases and the message latency increases. In Fig. 5-15 the dependency of the message latency on the vertical link rates (evenly distributed in the 3D NoC) for different traffic loads is shown. For each number of available vertical links, the message latency increases as the traffic load increases.

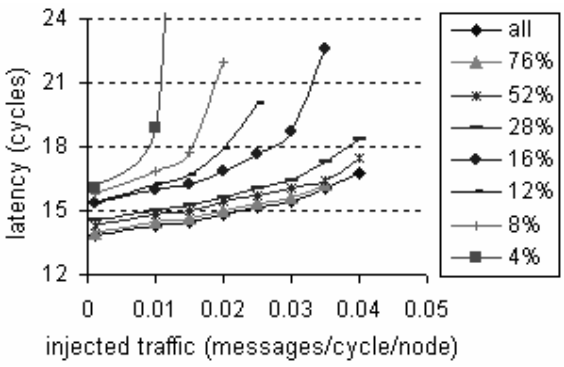
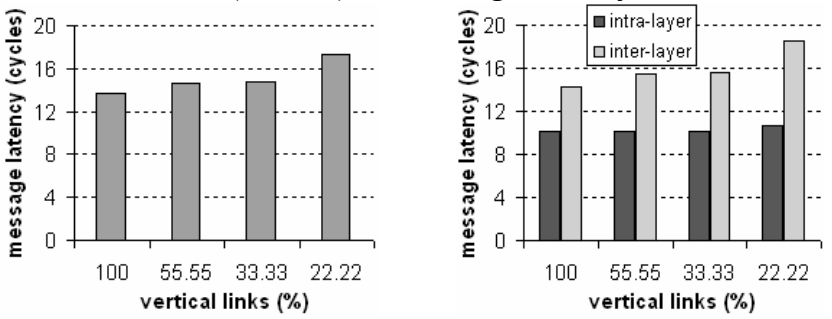


Fig. 5-15 Message latency for different number of vertical links and traffic loads

b) Vertical Links Impact on Inter-layer and Intra-layer Latency

A 6x6x3 mesh NoC (6 stacked layers, each of them having a 3x3 mesh topology) is considered for the next simulations. It has a maximum of 90 vertical unidirectional links (18 links between each of the five pairs of adjacent layers). The number of failed/missing vertical links can range from 0 (0%) up to 80 (88.88%); ten links have to be non-faulty, in order to ensure the connectivity between layers (assumption *A5-1*); two links are necessary between each pair of adjacent layers.

For a relatively low traffic load of 0.01 messages injected at each cycle by every node and each message having 20 flits, the message latency increases with the decrease of the number of available vertical links (that is the increase of the number of failed/inexistent vertical links), as shown in Fig. 5-16.a. However, even for a very small rate of available vertical links (22.22%), the message delivery is still carried out.



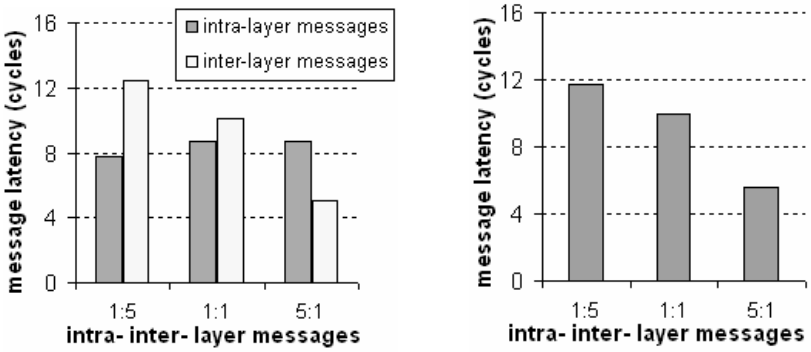
a. all messages b. intra- and inter-layer messages

Fig. 5-16 Message latency vs. number of failed vertical links

The latency considering separately the intra- and inter-layer messages is presented in Fig. 5-16.b. As the available vertical link rate decreases, the latency of intra-layer messages is not significantly affected. However, their number is small, as the destination is randomly chosen. As the NoC has 6 layers, the report of intra-layer : inter-layer messages is 1:5 (for *n* layers, it is 1:(*n*-1)). This explains also why the latency of all messages is closer to the latency of inter-layer messages. On the other hand, the latency of inter-layer messages presents a higher increase with the decrease of available vertical links rate. This is explained by the reduced number of available vertical links. In such a situation, all inter-layer messages compete for the same reduced set of vertical links. For a very small rate of available vertical links (22.22%), the intra-layer message latency also increases. This is due to the vertical messages that all wait in 2D layers before passing through the limited number of vertical links. Thus, they impede also the intra-layer traffic.

c) Inter-layer and Intra-layer Traffic Pattern

For the following simulations, a traffic of 0.004x20 (messages/cycle/node x flits) is injected in the 6x3x3 NoC. Only 44% of the vertical links are available. Different patterns of intra-inter-layer messages are considered. The results are depicted in Fig. 5-17.a and b.



a. Intra- and inter-layer message latencies b. All message latency

Fig. 5-17 Message latency for different traffic patterns

In the case of the 1:5 report of intra-inter layer messages, the latency of inter-layer messages is higher than in the case of intra-layer messages. This happens because the number of inter-layer messages is five times that of intra-layer messages and they all use the limited number of vertical links to reach their layer. The same tendency is maintained for the 1:1 traffic pattern, but the difference between the two latencies is smaller. For the 5:1 case, the latency of intra-layer messages surpasses that of the inter-layer ones. This is due to the fact that the 2D layers become congested because of the high number of intra-layer messages. The average latency of all messages independently of their type (intra- or inter-layer) is depicted in Fig. 5-17.b. The smallest latency is obtained when the number inter-layer messages is the smallest.

d) Impact of Number of Crossed Layers

The same topology with 44% available vertical links is considered for a detailed analysis of the latency of inter-layer messages. A uniform traffic of 0.01x20 (messages/cycle/node x flits) with an intra-:inter-layer message report of 1:5 is injected. The latency of inter-layer messages is ~15.5 cycles.

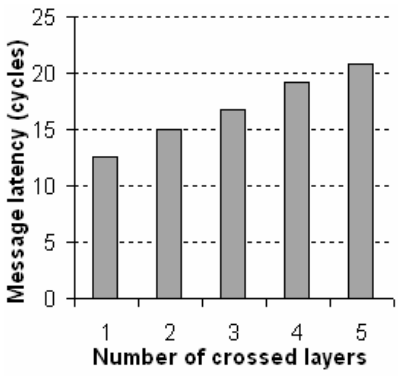


Fig. 5-18 Message latency for different number of crossed layers

The latency of messages considering the exact number of layers between their source and destination is presented in Fig. 5-18. The greater the number of taken layers, the highest the latency. However, the average latency of inter-layer messages (~15.5 cycles) is closer to the latencies for smaller number of crossed layers in Fig. 5-18. The way the destination of a message is chosen (randomly) during the traffic generation explains this situation, too. As the layers are stacked, the probability of smaller number of layers between the source and the destination of a message is higher than the probability of having a greater number of layers. For 6 stacked layers, for instance, for a layer distance of 2, eight pairs of layers are valid; while for a layer distance of 5 only two pairs of layers are valid (pair 1-6 and pair 6-1). Since the messages with smaller number of crossed layers are more numerous, they have more influence on the average then those with greater number of crossed layers.

e) Impact of Destination Layer

In the following part of this section, the latency of messages considering their destination layer is analyzed, for the same topology and traffic pattern as in the previous experiment. The simulation results are depicted in Fig. 5-19. As it can be observed, the latency of messages having the destination closer to the center of the layer stack is lower than in cases where the destination is closer to the ends of the stack.

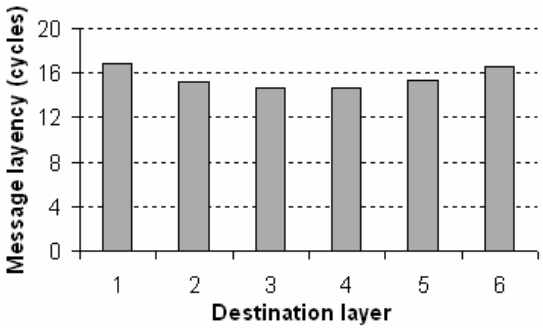
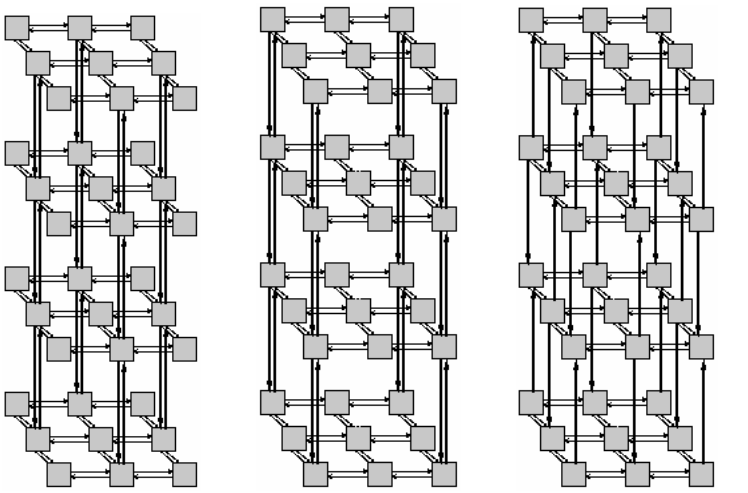


Fig. 5-19 Message latency for different destination layers

As shown previously, interlayer messages are prevailing for the chosen 3D stack and the traffic type. Several distinct vertical paths are available to reach the middle layers; that is paths from both directions. On the other hand, all messages arriving at a border layer (1 and 6 in this case) share the same vertical paths (for the same direction). Moreover, as these paths are relatively long, the probability that parts of them are busy is high. Thus, messages that will use them will be delayed more along their path. This is why the messages heading towards border layers have higher latencies.

f) Impact of Disposition of Vertical Links

With the following simulations we show that not only the number of available vertical links is important for the message latency, but also their disposition. Three different dispositions of vertical links are depicted in Fig. 5-20. All of them have 8 links between each pair of adjacent layers – 4 links for each vertical direction (44% of the total possible). For the first two dispositions (Fig. 5-20.a and b), each vertical node has a complete set of vertical links. In the third configuration, each vertical node is exclusively either a V_{To} or a V_{From} node.



a. middle V_{To} b. corners V_{To} c. different $V_{ToUp/Down}$

Fig. 5-20 Different vertical link dispositions

Simulations are done only for these three vertical link disposition configurations (as the exploration space can be huge), considering a 3D NoC formed of 6 layers (only four layers are depicted in the previous figure for clarity, but the same pattern can be repeated for several layers). A 0.004 x 20 (messages/cycle/node x flits) traffic is injected in the NoC by each node. The results are depicted in Fig. 5-21.

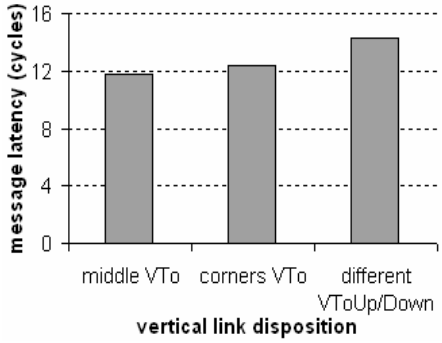


Fig. 5-21 Message latency for different vertical link dispositions

The message latency is the lowest for the first configuration and the highest for the last one. For the first two configurations, an inter-layer message must be routed horizontally only in the source and the destination layers, since at each arrival node in intermediary layers the vertical route can be directly continued from the current node. However, in the configuration in Fig. 5-20.a, the vertical links are more conveniently located: the distance from a node to its VNT is maximum 1 hop (in the second configuration, this is 2), which explains the highest latency for the second configuration. In the third configuration, the inter-layer messages are forced to actually route in each horizontal layer, as each vertical node is either a V_{To} or a V_{From}. In this case, the latency of intra-layer messages also increases, not only that of inter-layer ones, as the traffic load in horizontal layers is higher.

The impact of the number of vertical links on the message latency can vary depending not only on the traffic load, but also on the traffic pattern, i.e. inter-layer and intra-layer messages ratio, as it is shown in the following paragraphs.

In fact, the amount of available vertical links influences directly the inter-layer messages latency. The intra-layer latency is also influenced, but indirectly and only in

extreme conditions, for instance when the amount of inter-layer messages is very high compared to the number of available vertical links. In such cases, the inter-layer messages wait longer in 2D layers to pass through the vertical links and thus they can impede the flow of intra-layer messages. In our case, the source and destination of each message are randomly chosen with equal probability; therefore, for the 3 layer NoC, the probability of inter-layer messages is doubled with respect to the probability of intra-layer messages. In consequence, the influence of the number of available vertical link on the overall latency is quite high. However, the vertical links in our topology become a bottleneck in the NoC when their amount drops under 25% of the maximum possible (from the architectural point of view), as it can be observed in Fig. 5-15.

Not only the amount of available vertical links has an impact on the NoC performance, but also their pattern. The **pattern** of the vertical links involves both their **distribution** in the layer and their **continuity** from one layer to another. The latter determines the presence ratio of inter-layer messages in 2D layers. Regular patterns of vertical links were considered for the previous simulations. For the 3x5x5 topology with 52% available vertical links, six irregular vertical link patterns were also considered under a traffic load of 0.01 messages/cycle/node. In terms of their number, the vertical links are equally distributed between the two pairs of adjacent layers. Their distribution in each layer and their continuity from one pair of layers to the other are irregular. The message latency variations are of up to 5.5% (relatively to the minimal case).

g) Impact of Available Resources in 2D Layers

In the following part, we analyze the influence of the available resources in 2D layers on the message latency. A NoC having a 3x3x6 mesh topology (three stacked layers, L_0, L_1 and L_2, each having a 3x6 mesh topology) with an initial 50% vertical link rate (evenly distributed and continuous from one pair of layers to the next) is considered as the fault-free configuration. The chosen node failure rate is of 12.96% (7 nodes out of 54). However, different failure rates are considered for each layer. The failed nodes distribution in layers and their type are detailed in Table 5-2.

Table 5-2 Failed nodes number and type in each layer

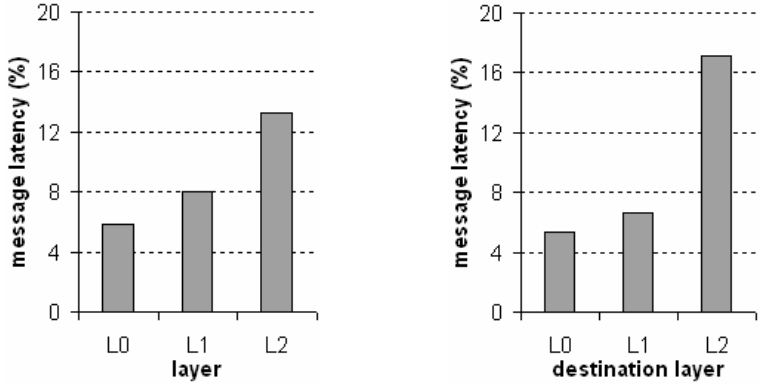
failures / layer	failure rate in layer (%)	total faulty nodes	2D faulty nodes	3D faulty nodes
L_0	5.55	1	1	0
L_1	11.11	2	2	0
L_2	22.22	4	2	2

Because of the failure of two 3D nodes in layer L_2, four links between L_1 and L_2 are also unusable (two for each direction). In the mean time, their corresponding 3D nodes in layer L_1 become exclusively 2D nodes after the failures.

The traffic injected is of 0.004 messages/cycle/node, each message having 20 flits. For an exact comparison, the same traffic is injected in the two NoC configurations (the fault-free NoC and the one with failures). For a fair comparison, only nodes that are not failed in the second NoC configuration are injecting traffic. The overall message latency increases with 9.8% after failures. More detailed latency results are presented in Fig. 5-22.

In Fig. 5-22.a, the latency of intra-layer messages is presented for each of the 3 horizontal layers. For example, in layer L_0, the latency is ~6% higher, because of the 2D

node failure in this layer. As expected, the latency of intra-layer messages increases with the failure rate in the layer (as presented in **Table 5-2**).



a. Intra-layer messages b. Destination layer

Fig. 5-22 Latency overheads (%) for 2D and 3D node failures

Results considering the messages by their destination layer are depicted in Fig. 5-22.b. For example, the latency of messages arriving to layer L_0 (from layers L_0, L_1 or L_2) is ~5.5% higher because of the failures in the NoC. The latency of the messages destinated to each layer also increases with the failure rate of the layer.

However, the layer L_2 is the destination the most affected by the failures. This is due not only to the higher node failure rate in the layer, but also because some 3D nodes are also failed. Thus, the layer can be reached with a higher latency. After the failures, the number of unusable vertical links from L_2 to L_1 is equal to those from L_1 to L_2. Still, the latency of messages destinated to L_1 does not increase so much as of those destinated to L_2. This can be explained by the position of the layers in the stack. In fact, the messages reaching L_1 come also from L_0 and L_1, not only from L_2. The failure rates in L_0 and L_1 are smaller and there are no failed vertical links between these two layers. On the other hand, messages from both L_0 and L_1 that reach L_2 have to pass through the reduced number of vertical links between L_1 and L_2.

5.7 Limitations

The main limitation of RILM consists in not dealing with partitioned layers. However, such a case is not likely to occur if the layer is homogeneous, i.e. the probability to have a sufficient failure rate and a specific pattern to partition the layer is quite high.

The other limitations of RILM for building an efficient 3D routing algorithm reside in the 2D routing algorithms that are used in 2D layers. For example, the overall 3D routing characteristics strongly depend on the 2D routing algorithms characteristics (fault tolerance, deadlock-freedom, congestion in 2D layers etc.). Several 2D routing algorithms achieve their fault tolerance property by defining unsafe nodes or zones around faulty nodes. Unsafe zones are avoided during routing in order to obviate the packet trap inside it. A 3D node may belong to an unsafe zone in its layer, created after the failure of some of its neighbors. In cases when unsafe nodes are completely deactivated and can not be destinations of messages, the vertical link(s) of unsafe 3D nodes can not be use any more. To avoid vertical link wasting, the 2D routing algorithm should be slightly modified so that to allow unsafe 3D nodes being final destinations in their layer.

A key point in RILM is the weight function to achieve adaptability to the application and system specifics. Congestions in a vertical link affect the 3D routing performance; therefore special care must be taken in calibrating the weight function.

Regarding the simulations, not all elements are properly taken into account; the propagation of a flit through a link is done in 1 cycle, for instance, but in reality it can take a few hardware clock cycles and can differ for horizontal and vertical links, depending on the actual technology. Even for the same type of links (horizontal/vertical), the transmission rate can be different if the number of actual wires in each link varies.

5.8 Conclusions

In this chapter we presented RILM, a method to extend to 3D stacked NoCs any 2D routing policy. RILM is designed for the case when the vertical connections of the NoC stack are incomplete. Already validated 2D routing policies can be used in 2D layers. The main characteristics of the routing algorithm for 3D NoC obtained by RILM are the following.

The 3D routing algorithm obtained by RILM tolerates a high number of multiple node and link failures. It can correctly find a route between any source and destination in the 3D NoC, if the latter is not partitioned. For the intra-layer routing, the tolerance degree depends on fault-tolerance properties of the 2D routing algorithms used in 2D layers. However, the failures of vertical links are tolerated, whether the 2D routing algorithms are fault-tolerant or not.

The algorithm locally reconfigures after failures; no central coordinator is necessary.

RILM ensures deadlock-freedom for the inter-layer routing. Besides, if the 2D routing algorithms in layers are deadlock-free, the whole 3D route is deadlock-free.

Adaptability to the application requirements and system characteristics is possible using the weight function.

RILM incurs a low overhead in terms of routing complexity and latency. It can be applied to any topologies of layers. RILM can be used to compose 3D routing algorithms for NoCs from simple or fault-tolerant 2D routing policies, even if they are different for different layers. The properties of the algorithm have been proven theoretically and validated by simulations in 3D NoCs with mesh layers.

Chapter 6. Recovery from Temporary Errors at Link and Transport Levels

This chapter presents fault-tolerant solutions for temporary faults in NoCs, acting both at data link and transport layers. Application layer is also involved, as the fault-tolerant techniques can be configured with respect to the QoS level of the transmitted data. A generic library of fault-tolerant modules implementing several types of EDC/ECC codes has been developed. The purpose of this work is to facilitate the choice of the most suitable fault-tolerant approach for the targeted level of fault-tolerance, depending on the QoS, and balancing it with the incurred cost in terms of extra- area, power consumption and latency.

6.1 Error-Detection and Error-Correction Codes

Many types of codes have been proposed and are used on VLSI circuits and communication systems. Error detection/correction codes (denoted by EDC and ECC, respectively) represent lower-cost solutions to tolerate errors, when compared to other classical fault-tolerant mechanisms such as error masking by triple modular redundancy (TMR) or the more general N-modular redundancy (NMR). The modular redundancy technique consists in replication of a module N times (N is chosen depending on the targeted fault-tolerance capability), supplemented by a voter. The voter architecture is the critical part of the design; therefore it should be very reliable.

A few definitions used in coding theory are given below.

Definition. The (Hamming) **weight** of a string (code word) is the number of symbols that are different from *zero*.

For binary codes, the weight is the number of 1's in the string.

Definition. The (Hamming) **distance** between two strings of equal length is the number of positions at which the corresponding symbols are different.

For binary strings *a* and *b* the Hamming distance is equal to the number of ones in *a* XOR *b*, i.e. the *weight* of *a* XOR *b*.

Definition. The (Hamming) **distance of a code** is the minimum distance between any two code words in the code.

The distance of a code defines its error detection and correction capability. A code with distance *d* can detect up to *d-1* errors and can correct up to $\lfloor (d-1)/2 \rfloor$ errors. The single-bit parity has a code distance of 2. Therefore, the single-bit parity code can detect one-bit errors, but can not correct them. To correct single-bit errors, the code must have a distance of 3. A code with distance 3 is called *single-error-correcting* (SEC) code. A code with distance 4 is called *single-error-correcting and double-error-detecting* (SEC-DED) code.

In general, detection/correction capability is achieved at the price of overheads in area, power and latency. For example, a single parity bit code has small area overhead, but detects only odd number of erroneous bits. Turbo codes, on the other hand, are more complex and generate more overhead, but achieve higher protection degree. Optimizations to reduce overheads exist, such as LDPC and Hsiao codes. CRC are interesting for their high capability of detecting burst (adjacent) errors.

Linear codes are largely used in fault-tolerant transmissions, given their efficient encoding and error detection and correction algorithms (i.e. using *error syndromes*). Linear codes include: parity, Hamming, Hsiao, and CRC codes. Our fault-tolerant schemes at link and transport layers make use of linear codes. Therefore, they are briefly presented in the following sections.

6.2 Linear Codes

Linear codes are a special class of block codes in which the modulo-2 sum of any code words is also a code word [LTW07].

6.2.1 Principle of Linear Codes

The principle of linear codes consists in adding redundant information to data, during *encoding* operation. Thus, the encoded data contains more bits than the original data. The

encoded data is transmitted through the communication channel and in case errors affect the encoded data during transmission, these are detected or corrected, by using the redundant information of the code. At receiver, the original data is extracted from the encoded data by *decoding*.

Let's denote the original data word by *d* (of length *k*) and the code word by *c* (of length *n*). Two **matrices** are used for linear codes:

- *Generator matrix*, $G_{k \times n}$, to obtain the code word from the data word:
  $c = d \times G$ (6-1)
- *Parity-check matrix*, $H_{(n-k) \times n}$, used to detect (and possibly correct) eventual errors. For any code word *c*, the following equality is true:
  $c \times H^T = 0$ (6-2)

The relation between *G* and *H* is:

$G \times H^T = 0$ (6-3)

The ***error syndrome*** is used for error detection and correction. Let's denote the received data word by *r*. This data is equal to the sent data, *c*, which may be affected by eventual errors that occurred during transmission:

$r = c + e$ (6-4)

where *e* represents the so-called *error vector* or *error pattern*.

By multiplying by H^T the equation (6-4), we obtain the error syndrome, *s*:

$s \equiv r \times H^T = c \times H^T + e \times H^T$ (6-5)

According to property (6-2), we obtain:

$s \equiv r \times H^T = e \times H^T$ (6-6)

Therefore, using *r* and *H*, the error syndrome can be computed at receiver.

When the received codeword is not affected by errors, *s* is 0. Otherwise, errors occurred. The error syndrome, *s*, is interpreted to identify the erroneous bits and correct them.

G and *H* matrixes can be mutated into equivalent codes by the following operations: column permutation and elementary row operations (replacing a row with a linear combination of rows) [Moo05].

G in the ***systematic*** form is:

$G = [I_k \mid P]$ (6-7)

where I_k represents the identity matrix, and corresponds to the original data bits. *P* represents the parity matrix; each column in *P* corresponds to a parity bit and indicates the bits checked by the respective parity bit. The corresponding *H* matrix is given by:

$H = [-P^T \mid I_{n-k}]$ (6-8)

In binary codes, $-P = P$.

There are several advantages of systematic codes, such as rapid and simple retrieval of original data and recomputation of only check bits at receiver.

For systematic codes, the error syndrome can be obtained by recomputing at receiver the check bits and then computing the difference (XOR) between the two sets of check bits (received and recomputed). An example is given below.

6.2.2 Example of Error Detection and Correction

Error detection and correction using the syndrome for systematic linear codes is exemplified in this subsection. Let's consider *G* and *H* matrices below (corresponding to Hamming code (7,4)):

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Suppose the data sent through the channel is d = [1 0 0 1].

The encoded word obtained by applying formula (6-1) is: c = [1 0 0 1 1 1 1]. The check bits added to the data are u = [1 1 1].

Consider the data is affected by the error pattern e = [0 0 0 1 0 0 0], i.e. the 4th bit is reversed. Thus, the receiver receives r = [1 0 0 0 1 1 1]. At receiver, the check bits for [1 0 0 0] are v = [0 0 0]. The syndrome is obtained as the difference (XOR) between u and v. Thus, s = [1 1 1]. This corresponds to the 4th column in matrix H. The correction is effectuated by reversing the value of the 4th bit of r.

6.2.3 Classes of Linear Codes

Polynomial codes represent a large class of linear block codes. The set of valid polynomial code words consists of those polynomials that are divisible by a given polynomial, called *generator polynomial*.

Cyclic codes are a *special type of polynomial codes*. A **cyclic code** is a *linear* (*n*, *k*) block code with the property that *every cyclic shift of a codeword results in another codeword*. Cyclic codes are used to detect *burst errors*. Burst errors are of particular interest because multi-bit errors tend to be clustered together. Noise sources tend to affect a contiguous set of bus lines in communication channels.

Cyclic redundancy check (CRC) is a very popular *error detecting* cyclic code implemented in many data transmission schemes. Other examples of cyclic codes are: *BCH* and *Reed-Solomon* [RS60] [KV01]. BCH codes are error correction codes with high Hamming distance. Reed-Solomon codes are a subset of BCH codes with particularly efficient structure.

Since the number of linear codes is very large, only a few classes of codes are detailed below. However, all linear codes can be implemented using G and H matrixes.

6.2.3.1 Parity Codes

This subsection dwells on parity codes used for error detection, such as simple and block parity.

Single-Bit Parity

A simple parity code (Fig. 6-1) consists in adding a single bit to the transmitted data bits, called *parity bit*, indicating whether the number of "1" bits in the data is even or odd. The encoding and error detection can be directly implemented with XOR gates.

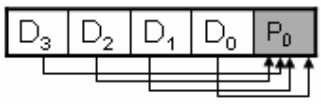


Fig. 6-1 Parity code

Error detection: If an odd number of bits are erroneous, being "0" instead of "1" or vice versa, the parity bit no longer reflects the correct number of "1"s. Error detection is

performed by comparing the parity bit and the recomputed parity of the data at the receiver module.

The main **advantage** of the parity code is the low overhead and high scalability: only a parity bit for any length of data. However, the larger the data size, the lower the detection capability of the code, because of the increased probability of error occurrences (if the same error probability is considered for each bit).

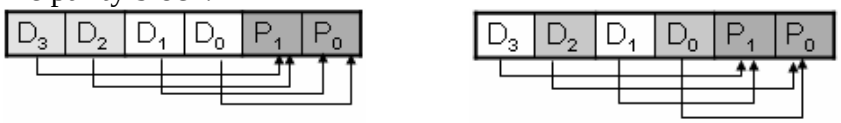
The main **drawback** of the parity code remains the even number of errors, as they are not detected.

Disjoint Block Parity

In order to improve the detection effectiveness of the code, several bits of parity are used. Data is grouped in blocks, and a parity bit checks each data block. This way, errors of *even* multiplicity are possibly divided into several errors of odd multiplicity. The respective parity bits indicate the errors in each block, increasing thus the detection capability of the code in these cases. The detection capability of *odd* multiplicity errors is still possible using the block parity, as the grouping of odd multiplicity errors implies the existence of at least one odd multiplicity sub-error (the sum of an odd number contains at least one odd number). Therefore, considering both even and odd multiplicity errors, the overall detection capability of the block parity code is higher than the simple parity code.

Parity bits can be applied to contiguous (Fig. 6-2.a) or interleaved blocks of data (Fig. 6-2.b).

In the *compact (or contiguous) parity block* approach (Fig. 6-2.a), adjacent bits belong to the same parity block.



a. Compact blocks b. Interleaved blocks

Fig. 6-2 Block parity

In the *interleaving parity blocks* approach (Fig. 6-2.b), adjacent bits belong to different parity blocks. This technique is the most effective solution for bus communication, where adjacent errors are more likely to occur. More bits of interleaving parity ensure better error cover, but at higher cost.

Another technique exploiting the localization of parity bits consists of storing these among data bits. This technique is used by certain *redundant array of inexpensive disks* (RAID), which enables the recovery of the data in the case of a failure of any disk [PGK00].

Overlapped Block Parity

In this type of code, the blocks (called also *groups*) are not disjoint, i.e. a bit belongs to several parity groups. A parity bit is assigned for each group; therefore, each bit is checked by several parity bits. Hamming codes belong to this family of codes.

6.2.3.2 Hamming Code

The Hamming [Moo05] and the Hsiao [CH84] [Hsi70] codes can be considered as parity codes, since they use overlapped parity blocks (groups), where a bit belongs to several parity groups (i.e. each bit is checked by several parity bits). On account of its redundant information, this approach also allows the correction of errors, not only their detection. In Fig. 6-3, an overlapped block parity example is presented, which is in fact the (7, 4)

Hamming code. The encoded word is on 7 bits: 4 are the original data bits and 3 represent the parity bits.

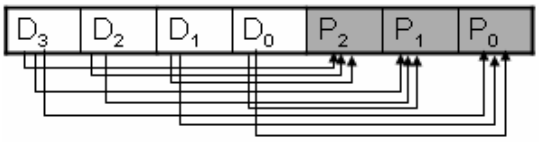


Fig. 6-3 Overlapped blocks parity. Hamming (7, 4)

The Hamming (7, 4) code can detect and correct single-bit errors; it is also called a SEC (*single error correction*) code. In the binary Hamming code of order r, the columns are all the none-zero binary vectors of length r. Each such column represents the binary form of an integer between 1 and $n = 2^r - 1$.

If an overall parity bit (bit 0) is included, then the code can also detect (but not correct) any double-bit errors. The code thus obtained is a SEC-DED code (single error correction and double error detection).

6.2.3.2.1 Hsiao Code

Hsiao code is an optimal SEC-DEC code invented by M.Y. Hsiao [Hsi70], derived from the more general Hamming codes. Hsiao code is a minimal odd-weight-column, i.e. the weight of each column is odd and the number of '1's in the check matrix is minimal. The minimum number of "1"s means minimal power required. Compared with Hamming codes, Hsiao codes provide **improvements** in speed, cost and reliability in the decoding logic.

6.2.3.2.2 Low-Density Parity-Check (LDPC) Code

LPDC codes were first proposed by R.G. Gallager in 1963 and rediscovered over 30 years later. The name comes from the characteristic of their parity-check matrix, which contains only a few 1's in comparison to the amount of 0's. The main **advantage** of LDPC codes is that they provide a performance which is very close to the capacity of a lot of different channels and linear time complex algorithms for decoding. Furthermore, they are suited for implementations that make heavy use of parallelism.

6.2.3.3 Cyclic Redundancy Check (CRC) Code

Cyclic redundancy check, CRC, (or *polynomial code checksum*) codes are specifically designed to protect against common types of errors in communication channels, where they can provide quick and reasonable assurance of the integrity of messages delivered.

The CRC was invented by W.W. Peterson and published in 1961 [PB61]. An important reason for the popularity of CRCs for detecting the accidental alteration of data is their efficiency guarantee. Typically, an *n*-bit CRC, applied to a data block of arbitrary length, will detect any single error burst not longer than *n* bits and will detect a fraction $1 - 2^{-n}$ of all longer error bursts. Errors in both data transmission channels and magnetic storage media tend to be distributed non-randomly (i.e. are "bursty"), making CRCs' properties more useful than alternative schemes such as multiple parity checks.

A CRC code is formed using a *generator polynomial*, $g(x)$, which is also called the *code generator*. Its degree is equal to the number of check bits, i.e. n-k:

$$g(x) = g_{n-k}.x^{n-k} + \ldots + g_2.x^2 + g_1.x + g_0 \qquad (6\text{-}9)$$

A codeword, $c(x)$, is formed by Galois (modulo 2) multiplication of the message, $m(x)$, with the generator, $g(x)$:

$$c(x) = m(x).g(x) \quad (6\text{-}10)$$

For systematic CRC codes, the code word can be obtained as follows:

$$c(x) = m(x).x^{n-k} + r(x) \quad (6\text{-}11)$$

where $r(x)$ = remainder of $(m(x).x^{n-k}) / g(x)$

This involves Galois (modulo 2) division, which can be performed using *linear feedback shift register* (LFSR), which is a compact circuit.

As all linear codes, CRC code has a corresponding *G*-matrix and *H*-matrix. Each row of the *G*-matrix is simply a shifted version of the generator polynomial.

The generator polynomial is the most important part of the CRC. It must be chosen to maximize the error detection capabilities. Several lengths are most commonly used and several generator polynomials are used for different standards.

Observation: *Single-bit parity* is a special case of cyclic redundancy check (CRC), where the 1-bit CRC is generated by the polynomial $x+1$.

6.3 Error Recovery Mechanisms

Based on error detection and correction codes, two main *error recovery mechanisms* can be used:

- *Error correction*, using error correction codes;
- *Data retransmission*, using error detection codes.

In NoCs, these can be applied:

- *Switch-to-switch* (*S2S*) or at *link level*, i.e. at each hop along the route of data;
- *End-to-end* (*E2E*) or at *transport level*, i.e. between the *network interfaces* (*NIs*) of the source and destination nodes.

The following tables resume the characteristics of the basic error recovery mechanisms (correction and retransmission) applied S2S or E2E.

S2S schemes deal with the errors cumulated along the path. In E2E schemes, error multiplicity may surpass the EDC/ECC capability (Table 6-1). However, a higher price is paid for the S2S implementation, in terms of area and power overhead, since the error control blocks must be added for each pair of neighbor nodes. In E2E schemes, these are present only at NIs. The retransmission buffer width can be narrower for S2S schemes, since buffers store original data (vs. encoded data in E2E schemes).

Table 6-1 Switch-to-switch vs. end-to-end error control effectiveness

Criterion/scheme	**S2S**	**E2E**
Deal with errors cumulated along the path	*yes*	*no*
Area and power overhead	*higher*	*smaller*
Intermediate buffer width	*narrower*	*wider*

The latency of error recovery mechanisms in the four basic cases of error recovery is presented in Table 6-2. A S2S retransmission takes 3 clock cycles, while the S2S error correction takes only 1 cycle. The correction has the same latency, even for the E2E case with any number of hops, but it does not deal with errors cumulated along the path. The latency of E2E retransmission can not be simply estimated, as it depends on several factors, like the length of the path between the two end nodes and the traffic along this path, for both the retransmission request and the data retransmission itself.

Table 6-2 Error recovery mechanisms latency overhead

Latency Overhead	**S2S**	**E2E**
Retransmission	*3 clock cycles / link*	*depends on the path, traffic load etc.*
Correction	*1 clock cycle / link*	*1 clock cycle*

Error correction codes are more complex than error detection codes, and thus, their corresponding costs are higher. On the other hand, the retransmission implies the existence of buffers to store data for future possible retransmissions. Thus, the overall area for retransmission is higher than in the case of correction (Table 6-3). However, by paying the retransmission buffer price, a higher effectiveness of the fault-tolerant scheme is achieved, since recovery is possible also from error patterns that can not be corrected by the error correction code.

Table 6-3 Correction vs. retransmission area overhead

Scheme	Area overhead
Retransmission	*higher*
Correction	*lower*

The effectiveness of the four basic error recovery schemes is synthesized in Table 6-4. When errors are likely to occur uniformly distributed along the path, S2S schemes are more effective, since they treat each error as it occurs. S2S error correction deals with errors of lower multiplicity (limited by the error correction code capability), uniformly distributed along the path. On the other hand, S2S retransmission is more effective, especially when there are many errors uniformly distributed along the path, which can not be treated by the error correction code.

Table 6-4 Correction vs. retransmission effectiveness

Efficiency	**S2S**	**E2E**
Retrans-mission	- *<u>High</u> multiplicity errors* - *Errors <u>uniformly</u> distributed along the path*	- *<u>High</u> multiplicity errors* - *Errors <u>not uniformly</u> distributed along the path*
Correction	- *<u>Low</u> multiplicity errors* - *Errors <u>uniformly</u> distributed along the path*	- *<u>Low</u> multiplicity errors* - *Errors <u>not uniformly</u> distributed along the path*

If errors are not uniformly distributed along the path, S2S error control is useless. To gain in latency, E2E schemes are preferred. Like in the case of S2S schemes, E2E retransmission can deal with several error patterns than the correction approach. This limitation is given by the error detection/correction capability of the selected code.

6.4 Error Control Schemes

Two main error control schemes correspond to the basic error recovery mechanism: error correction and data retransmission. In many cases, some types of errors are more likely to occur, while other types (usually those with higher multiplicity and irregular pattern) occur more rarely. Therefore, a trade-off between the effectiveness of the fault-tolerant

scheme and its cost must be considered. The solution to this issue is a *hybrid* scheme, which combines error correction with retransmission.

Considering applications with different QoS, we propose the *selective* error control scheme. The *selective* scheme detects errors and possibly corrects them. In this scheme, the correction is made based on the side-band information of data items (flits) send over the link. This information is added by the network interface, under the control of the applications or the drivers of the SoC IP.

The characteristics of the error control schemes mentioned above are synthesized in the following lines:

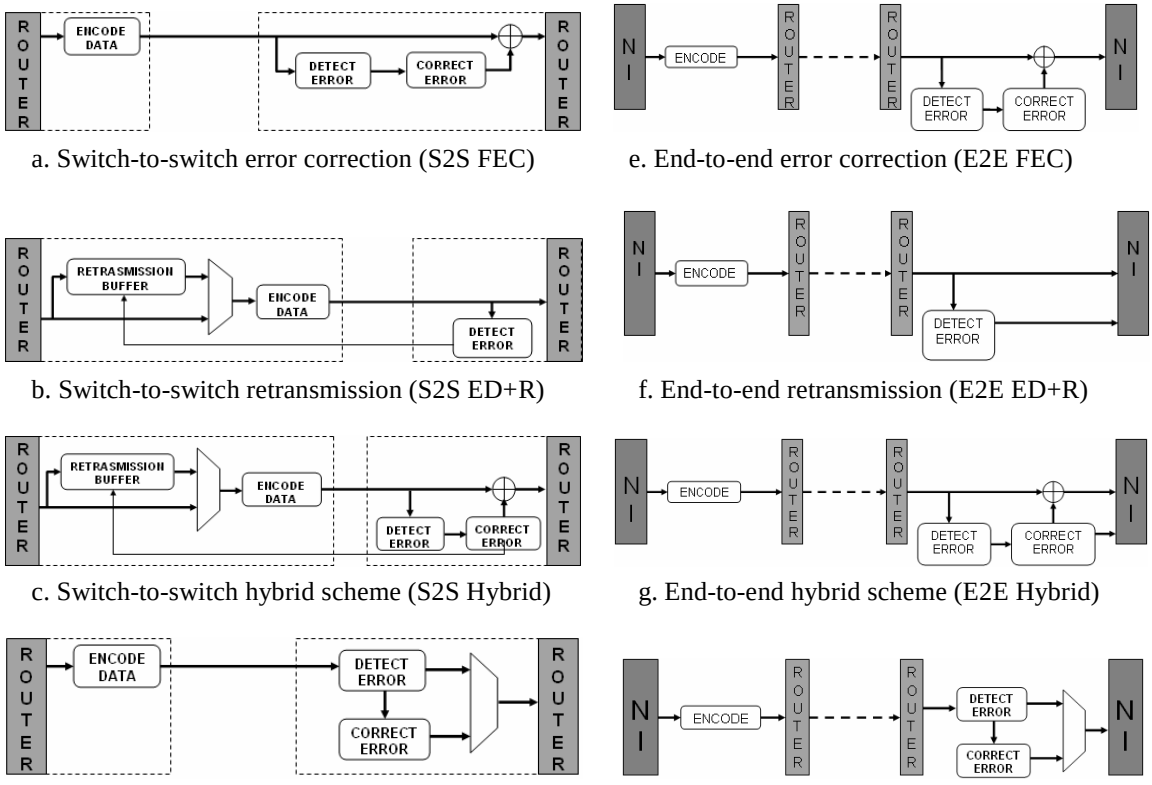
◊ *Forward Error Correction* (*FEC*): received data is always corrected (only error correction codes are used: Hamming SEC/DED, Hsiao).

◊ *Error Detection and Retransmission* (*ED+R*): resend data when errors are detected (error detection codes are used: parity, Hamming).

◊ *Hybrid*: correct as many errors as possible and alternatively request retransmission to improve error correction capability (error correction codes are used).

◊ *Selective*: correct errors only when explicitly specified by upper levels (error correction codes are used).

All these schemes can be applied either between each pair of adjacent switches (or routers) or only between the source and the destination of each messages, as depicted in Fig. 6-4. *Switch-to-switch* (*S2S*) schemes are depicted in Fig. 6-4.a-d and *end-to-end* (*E2E*) schemes, in Fig. 6-4.e-h.



a. Switch-to-switch error correction (S2S FEC)

e. End-to-end error correction (E2E FEC)

b. Switch-to-switch retransmission (S2S ED+R)

f. End-to-end retransmission (E2E ED+R)

c. Switch-to-switch hybrid scheme (S2S Hybrid)

g. End-to-end hybrid scheme (E2E Hybrid)

d. Switch-to-switch selective scheme (S2S Selective)

h. End-to-end selective scheme (E2E Selective)

Fig. 6-4 Error control schemes

- In *S2S FEC* scheme (Fig. 6-4.a), the modules added to the simple link between two routers/switches are: the encoder (at the sender end of the link), the detection and the correction modules (at the receiver end).

- In Fig. 6-4.b, the *S2S ED+R* scheme is depicted. The encoding module is also present at the sender part, and it is complemented by a retransmission buffer. When new data is sent, it is also stored in the retransmission buffer. In case of retransmission request, the data to be sent is taken from the retransmission buffer. At receiver, a detection module has the role of detecting the error and asking for data retransmission in case of errors. Thus, the communication channel is complemented with the retransmission request, in the reverse direction.
- The *S2S hybrid* scheme combining the error correction and the retransmission is presented in Fig. 6-4.c. Besides the error detection module, the hybrid scheme contains modules for both retransmission and error correction. In case of errors, the *hybrid* scheme performs correction whenever it is possible and retransmission otherwise. Since retransmission is slower than correction, the *hybrid* scheme is faster than pure retransmission. The adaptability and gain in latency of the *hybrid* scheme are however paid in area overhead.
- In Fig. 6-4.d, the *S2S selective* error correction scheme is shown. Data (*flits*) sent on wires are encoded before the data leaves the upstream interface logic.

Error detection and possible correction are implemented before data arrives in the downstream interface (Fig. 6-4.d). When errors are detected, the side-band information sent on control lines indicates whether or not correction should be made. If correction is not necessary, because either there are no detected errors or detected errors will be corrected by upper layers, flits arrive at the downstream interface with the delay induced by detection. When detected errors must be corrected, a supplementary delay is added by the correction operation.

On the outputs of detection and correction modules of the *selective* scheme (Fig. 6-4.d) there are uncorrected and corrected data, respectively. These outputs are multiplexed before data arrives at the downstream interface. The multiplexer is commanded by a flip flop that is set by a correction decision and it is reset when transmission ends. When the flip-flop is set, the output of the correction module is forwarded to the downstream interface. In the other case, the uncorrected data at the output of the detection module is transmitted. Therefore, once a flit is corrected, all subsequent flits are delayed with an extra clock cycle and bypassing the correction stage becomes possible after transmission ends. Thus, the *selective* scheme has a latency overhead similar with FEC when correction is necessary, and lower otherwise.

E2E schemes (Fig. 6-4.e-h) operate by the same principle as switch-to-switch schemes, but only from end nodes (i.e. between the source and the destination of messages). In the end-to-end schemes, the same modules are still present at each router connected to a *network interface* (*NI*), but they do not operate unless they belong to a source or a destination node of the current message, reducing thus the message latency. For the retransmission requests, unlike in the S2S cases, dedicated wires are not to be added in the NoC. Instead of the hardware implementation, the retransmission request is realized by messages traveling through the NoC.

In all error control schemes, the **control lines** containing critical information are protected with triple modular redundancy (**TMR**) and majority voting.

For error **detection**, single and double error detection codes are used. For **correction**, codes with single error correction capability are used.

To deal with **multiple errors**, block and interleaving techniques applied to these simple capacity codes are used to obtain codes with longer Hamming distance. To enhance the detection capability of a scheme, block and interleaving techniques applied to single or double error detection codes can be used (as detailed in subsection 6.2.3.1,

for parity codes). The same techniques can be used for correction. For example, to correct two errors in a flit, the flit is divided into two groups (depending on the expected error pattern) and each group is protected by the SEC Hamming code.

More groups increase the correction capabilities of the code. When data is split in *m* groups, all multiple error patterns of up to *m* errors can be corrected, if the errors are distributed such that there is at most one error per group. The way the groups are formed also influences the error distribution in the patterns. For example, crosstalk induced multiple errors usually affect adjacent wires and interleaving SEC codes (i.e. adjacent wires are in different groups) correct all these errors.

Nevertheless, any other type of higher distance codes can also be used instead of block/interleaving of simple codes.

6.5 Linear Code Library and Simulation Environment

Based on the common properties of linear codes, a code library was built. It makes use of a *G* and *H* matrixes database for several types and sizes of codes. However, other codes can be added in the library by simply appending their *G* and *H* matrixes to database.

For any linear code, encoding, error detection and correction modules are automatically generated, using matrixes *G* and *H*, and linear code formulas (see section 6.2). Formula 6-1 is used to generate the encoder. The error syndrome is computed with formula 6-6. Then, the syndrome is used to correct the error as exemplified in subsection 6.2.2.

The fault-tolerant link generation is depicted in Fig. 6-5. For each data size, the code type must be specified (*code ID*, *n*, *k*), as well as the number of groups (for block/interleaving coding). This information is used to select the appropriate matrixes from the database. Triplication and voting is used to generate fault-tolerant control signals.

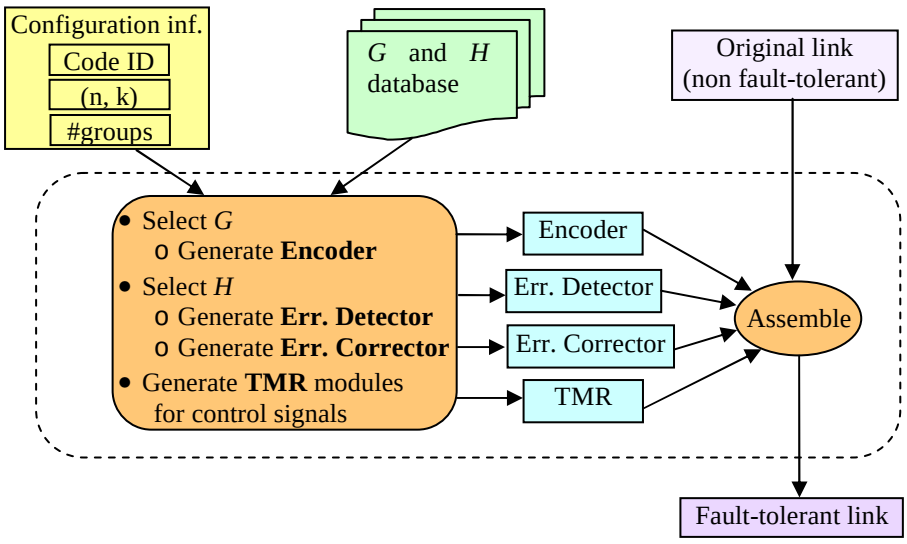


Fig. 6-5 Fault-tolerant link generation

The fault-tolerant link is obtained from the original link by adding in the upstream and downstream interfaces the fault-tolerant modules and replacing the control wires with their fault-tolerant counterparts.

For dynamic link simulations, a **traffic generator** and a **traffic sink** are connected the link under evaluation. The traffic generator generates burst traffic, i.e. a flit is injected in the link at each clock cycle. The traffic sink reads flits from the link as soon as they are available.

6.6 Case Study

6.6.1 Spidergon STNoC Link

3D Spidergon STNoC is the extension of the Spidergon STNoC [CGL+08] communication platform to stacked 3D integration technologies. The building blocks of Spidergon STNoC are the network interfaces, routers and links.

The network interface (NI) is the access point to the NoC, converting the messages generated by the IP or the subsystem connected to it into packets that will be transported through the network. The NI hides network-dependent aspects from the transport layer, allowing IP blocks to be re-used without further modification no matter how the NoC architecture subsequently evolves. The NI is also responsible for frequency and data size conversions between the IP/subsystem and the NoC. The Spidergon STNoC router is responsible for flit transmission, using wormhole routing protocol. The STNoC router is designed to support ST's proprietary Spidergon topology; therefore it is capable of routing packets via three different links: left, right and across.

STNoC link is responsible for the connections between routers and between routers and NIs. Spidergon STNoC link is a synchronous, parallel and unidirectional link [CGL+08]. Two such links connect two neighboring routers (possible in neighboring layers for 3D NoCs) or routers and the network interfaces. STNoC links can be used both in 2D NoCs as well as in 3D NoCs, as vertical links between 2D layers of the stack. In Fig. 6-6, the structure of the synchronous STNoC link is depicted.

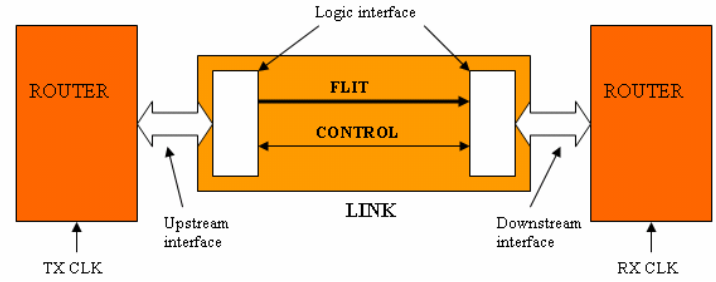


Fig. 6-6 Structure of synchronous STNoC Link

The link consists of two logic interfaces (upstream and downstream) and two sets of wires (data and control). The link uses the credit based flow control mechanism to transmit flits. Signals sent on control wires specify various properties of the transmitted flit (side-band information) such as its relative position in the packet (i.e. header, payload or final). The control wires also carry the signals necessary for credit based flow control (i.e. valid and credit).

6.6.2 Simulation Results

In this section we analyze the overhead of the error control schemes applied to the Spidergon STNoC link, in terms of area, power and wire count.

6.6.2.1 Impact on Latency

In order to assess the performance degradation of the link, the latency overhead on the link is measured from the moment the flit leaves the upstream interface to the moment when it arrives at the downstream interface. For evaluation, one unidirectional vertical link is considered and uniform traffic patterns are injected at a rate of one flit / cycle (i.e. burst traffic).

The latency overhead of the *FEC* scheme is constant, i.e. 2 cycles, one for error detection and the other for error correction. For the *ED+R* scheme, the minimum latency overhead is 1 and it increases depending on the retransmission rate. For the *hybrid* scheme, the latency strongly depends on the error pattern and rate.

The *selective* error control scheme corrects errors based on the side-band information of the transmitted data. Thus, unnecessary delays are avoided, such as delays caused by error correction when no errors are detected or when non-critical errors are detected. Therefore, the latency of data transmitted on the link depends on the transmission error rate. In Fig. 6-7 the data latency overhead is represented for increasing error rates. For this simulation, we suppose that all errors that occur must be corrected.

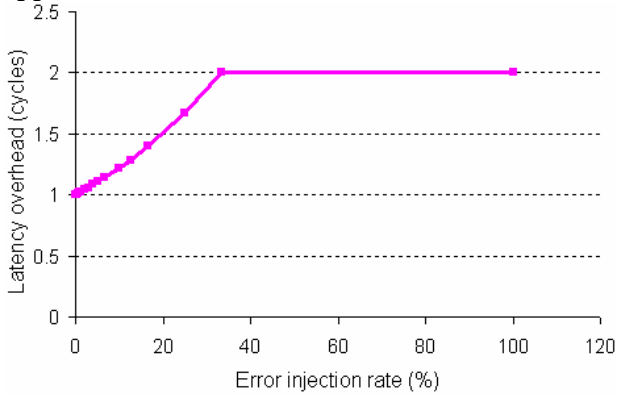


Fig. 6-7 Selective error correction scheme latency overhead

For relatively high error rates, the probability of needing correction early in the burst increases. This causes an increase of the latency overhead to two clock cycles for all subsequent flits, as bypassing the error correction stage is possible only after the transmission ends.

6.6.2.2 Impact on Area and Power

The improved reliability of communication on the synchronous Spidergon STNoC link comes with extra cost in silicon: area and dissipated power due to additional modules and wire count. To evaluate this cost, the RTL model of the link with improved reliability is synthesized in the 65 nm process from ST Microelectronics. The link is configured with two virtual channels, each with its own input buffer, and different flit sizes.

The area and power overheads for the four error control schemes (*FEC*, *ED+R*, *hybrid* and *selective*) are shown in Fig. 6-8. These overheads are induced by the fault-tolerant modules: encoder, decoder, corrector, retransmission mechanism and buffers and selection logic. The link is configured for 64 bits wide and 3 block/interleaving configurations are considered: 1 group of 64 bits, 2 groups of 32 de bits and 4 groups of 16 bits, for different multiplicities of expected errors. Errors of higher multiplicity can be treated with several codes, if erroneous bits are in different groups. This gain requires however

small area and power overhead (Fig. 6-8). Parity code is used in the *ED+R* scheme and SEC-DED Hamming code for the others.

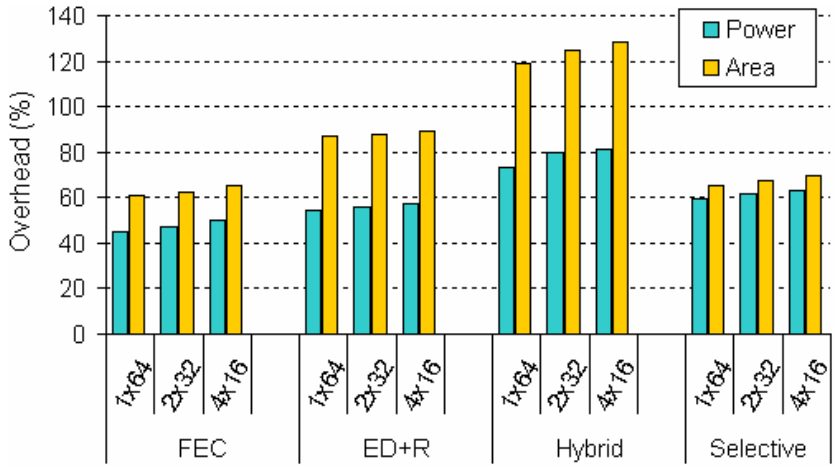


Fig. 6-8 Power and area overheads of switch-to-switch error control schemes

The *hybrid* scheme has the highest overheads (Fig. 6-8), as it contains modules for both error correction and retransmission. It is followed by the *ED+R* scheme, due to the retransmission buffers it also contains. *FEC* has the smallest overheads. The *selective* scheme has small area overhead, comparable to that of the *FEC* scheme. Regarding the power, it is higher, closer to that of the *ED+R* scheme. This is explained by the presence of the selection logic.

The overheads of the *selective* scheme are represented in Fig. 6-9, for different configurations of the link and coding schemes.

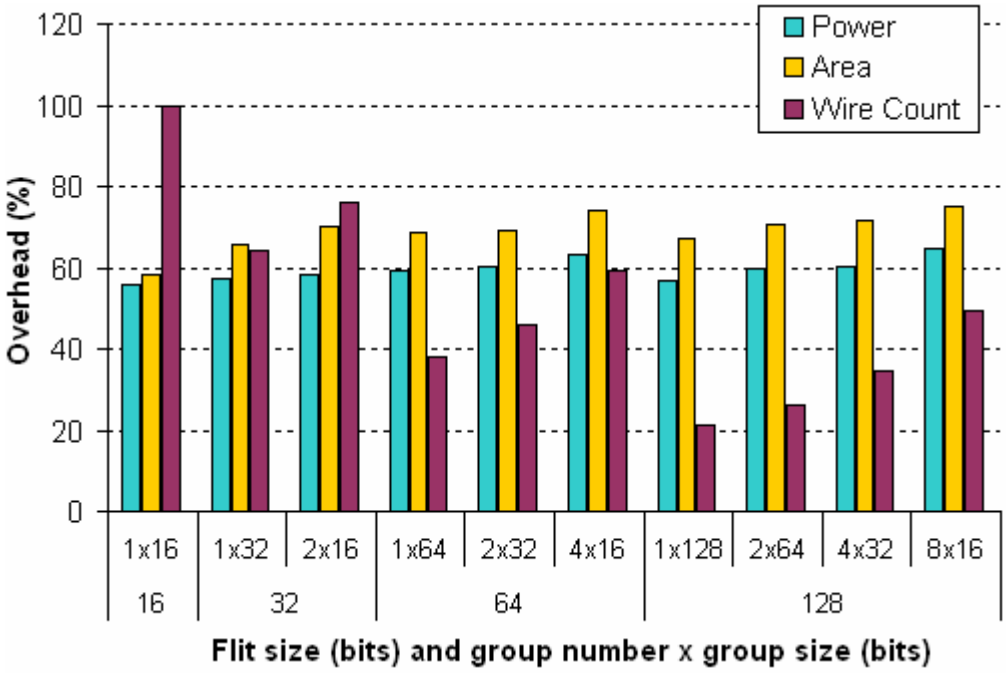


Fig. 6-9 Area, dissipated power and wire count overhead for different link configurations

The wire count is an important metric especially when the link is a vertical one in a 3D NoC, because the vertical bundles have large footprints on layers. In current technology TSVs have a pitch of tens of µm and 30 64-bit links have a total footprint of 0.7

μm^2, equivalent the area of an IP block. The wire count overhead gives an approximation of the area penalty due to the increased error correction capabilities (i.e. extra control bits). For example, 128 bits of data can be protected with single error correction capabilities with 9 additional wires, whereas protecting 128 wires with multiple (up to eight) error correction capabilities requires 48 wires (6 extra wires for each 16 bits). In the proposed scheme the control lines are protected using triple modular redundancy (i.e. for each signal there are three wires). In the analyzed configurations of the link the side-band information consists of six bits and four signals for the credit based protocol. These signals require 20 extra wires, independently of the flit size or coding.

The proposed scheme increases the round trip delay latency of the credits in the link as encoding, error detection and error correction each require one clock cycle and, in order to maximize throughput, the input buffers must be resized. The increase in flit size partially masks the increase in area incurred by the coding scheme. Larger flits mean wider buffers in the synchronous link and, in the same time, larger error control modules. In both cases the increase in area follows a similar trend and the variations in the area overhead stay in the same range for the analyzed configurations.

The dissipated power is determined by the synthesis tool for a clock period of 1 ns (i.e. 1 GHz). The dissipated power overhead exhibits similar behavior to that of the area overhead. However, when the flit size increases considerably, the dissipated power overhead on the link is considerably higher than in the case of the error correction scheme. The net result is a smaller overhead of the dissipated power for relative large flits.

The hardware analysis of the proposed scheme shows that the area and dissipated power overheads for the Spidergon STNoC synchronous link have small variations for small to medium sized flits. The wire count depends on number of groups in the coding scheme and impacts the footprint of the vertical interconnects as more groups mean more vertical wires. In the same time, more groups increase the error correction capabilities and, as the analysis shows, with relative small area and power penalties.

The area overhead of the selective error correction scheme is also evaluated relatively to the NoC. A 2x4x4 3D NoC (2 layers, each with a 4x4 mesh topology) is used for this evaluation. Three types of protection are considered: switch-to-switch, end-to end and on vertical links only (series VL in the graphic). The results are depicted in figure Fig. 6-10.

When compared to the area overhead for a single link (previous figure Fig. 6-9), lower overheads are obtained at NoC-level. Among the three configurations, the S2S implementation is the most area consuming and it is recommended for high error rate. Then, the E2E protection needs only less than 20% area overhead. The third configuration is particularly interesting to protect the vertical links from the errors that occur in TSVs. Complete vertical connectivity is considered in this estimation and all the vertical links in the NoC are protected. The area overhead is less than 10%. Moreover, the overheads at the system-based NoC level (including also the processing cores and the network interfaces) are even smaller. Therefore, even if at the link level the overhead is quite important, from the system point of view, they are small or even negligible in certain cases.

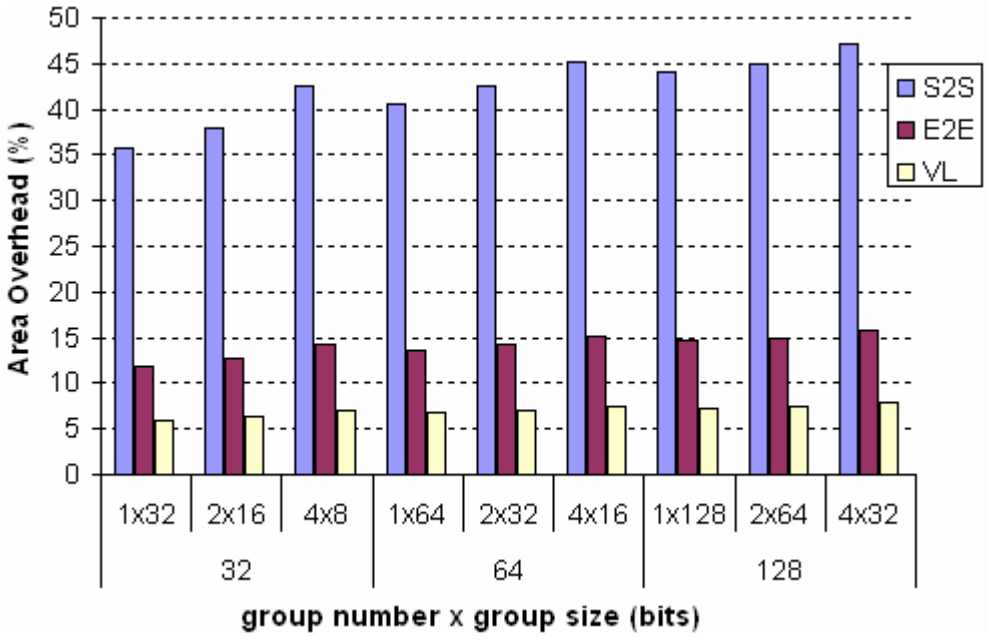


Fig. 6-10 Area overhead at NoC for S2S, E2E and Vertical Link only (VL) error protection

Observation: In this experiment (Fig. 6-10), the TSVs are considered with the same characteristics as the 2D links. More accurate results will be obtained using real characteristics of vertical links.

6.7 Conclusions

This chapter presents fault-tolerant schemes based on error detection and correction coding. Besides classical schemes (*FEC*, *ED+R* and *hybrid*), we developed a *selective* correction scheme. In the *selective* scheme, error correction phase is skipped when possible, based on the side-band information transmitted along with the data. This information is set at higher levels of the communication protocol stack, like application and transport levels. If necessary, uncorrected data is handled at these levels; for instance by end-to-end retransmission or at application level. Hence, a gain in latency is achieved for guaranteed throughput applications. For packets of high reliability applications, the correction phase is not skipped and errors accumulated along the path are treated at each hop. In this manner, a latency-reliability trade-off is attained, responding to heterogeneous application QoS requirements that can exist in a NoC. The *selective* scheme incurs low overheads in area and power, comparable to those of simple *FEC* schemes.

A generic library of linear codes has also been developed. These codes can be applied on different length of data words, as well as on blocks or interleaved groups, depending on the proposed level of fault-tolerance.

Chapter 7. Conclusions and Future Work

Modern chips are more and more complex and heterogeneous; they have different application QoS requirements and may be done in 3D integration technology. In the mean time, SoCs are prone to failures caused by transient, intermittent and permanent faults. These faults can originate in the shrunk dimensions of the electronic components, generated during fabrication or in the life-time, combined with operation-life environmental factors (radiations, interferences etc.).

The overall objective of this thesis is to address fault-tolerant methods that take into account the complexity of modern chips and their heterogeneity, in order to minimize the fault-tolerance overhead and achieve scalable solutions. Many fault-tolerant approaches have been proposed in the past, at all levels of the communication protocol stack and at all components and subcomponents of NoCs. However, this thesis is not an overview of these fault-tolerant methods, but focuses on three aspects: checkpoint and rollback, fault-tolerant routing and fault-tolerant transmissions. The solutions proposed in the thesis respond to the general question formulated in the second chapter:

- **Question**: How to achieve acceptable reliability levels with reduced impact on the QoS requirements by offering adaptability to the application specifics (data integrity, guranteed throughput, best effort)?
  Answer: The answer to this question is as general as the question. The answer can be composed by gathering the answers to the questions addressing each fault-tolerant mechanism. However, collaboration between different fault-tolerant mechanisms, at the same or at different levels, can significantly contribute in adapting the system reliability according to the QoS requirements, while minimizing the costs in latency, power consumption and area.

Checkpoint and rollback technique was originally proposed to rapidly recover from errors or failures of the application that are caused by transient or intermittent faults. The technique can also be useful in case of NoC node failures, to migrate and resume the corresponding tasks on other non-faulty nodes. A rapid restoration of the application can thus be achieved. The checkpoint and rollback technique has been largely used in macro-networks (e.g. distributed systems), but not yet in NoCs, to the best of our knowledge.

- **Question**: How the checkpoint and rollback mechanism impacts on the NoC performance can be evaluated and how its scalability can be improved?
  Answer: In order to evaluate the performances impact of this technique on NoCs, we developed different checkpoint and rollback approaches based on an abstract model of the application. Improvements of the basic protocols were also proposed to increase their scalability and minimize their impact on the application performance, considering the application specifics (such as critical or real-time tasks and communication partitioning). All protocols are topology independent. However, a routing mechanism adapted to the topology specifics can significantly contribute to improving the protocol scalability. By applying the scalability improvements at application and routing level to the coordinated checkpointing, up to two orders of magnitude reductions in latency and memory overhead are achieved for systems with 256 nodes. Significant reductions of the checkpoint duration (up to 5%) are achieved by exploiting the communication partitioning. The error rate that can be tolerated with the blocking coordinated checkpointing approach is several times higher (2 to 6 in our simulations, depending on the traffic load) than in the non-blocking one. A unified approach of blocking – non-blocking checkpoint is proposed, which exploits the different QoS requirements of applications.

3D NoC architectures exploit both the advantages of NoCs and of 3D integration with stacked layers. A 3D topology with partial vertical connectivity is an efficient solution to the utilization of a reduced number of TSVs and that adapts to the heterogeneity and the irregularities of actual and future 3D systems.

- **Question**: How to route messages in 3D NoCs with partial vertical connectivity and possibly different layer topologies, with minimal design effort, overhead and time-to-market, while ensuring an optimal routing inside each layer?
  Answer: RILM, an inter-layer routing algorithm for 3D NoCs with incomplete vertical connectivity and possibly different regular or irregular topologies in horizontal 2D layers has been proposed in this thesis. Already validated and adapted algorithms for 2D topologies can be used in 2D layers with minimal overhead (only a few supplementary fields must be kept at each node to enable the 3D routing). Besides of dealing with the vertical irregularities of 3D NoC topologies, the routing algorithm considers failures of routers and links of the NoC. In fact, missing and failed vertical links are treated in the same manner. RILM permits local reconfiguration in case of failures. (Re)joining of repaired or spare vertical links and routers is also possible using the same reconfiguration mechanism. Adaptability to different QoS requirements (considering for example the NoC actual traffic loads or vertical link bandwidth or throughput) is possible through a customizable function attached to the routing algorithm. The routing mechanism can scale to any number of layers in the 3D stack and can tolerate a high number of failures: only two vertical links (one to up and one to down) between adjacent layers are sufficient to ensure the routing, as long as the layers are not partitioned.

Beside permanent faults, transmissions in VDSM are affected by transient and intermittent faults.

- **Question**: How to deal with transient and intermittent faults in NoC links?
  Answer: The fault-tolerant scheme proposed in this thesis is based on error detection and correction codes. In order to respond to the application QoS heterogeneity, information from application or transport level is exploited for reliability-latency trade-off. More precisely, for guaranteed-throughput applications, the correction phase is omitted and a gain in latency overhead is achieved. Possible errors produced along the path are treated at transport and application level. On the other hand, for high reliability applications, the correction can be done at each hop in the path. Thus, errors can not accumulate along the path and compromise thus the data integrity. For best-effort applications, correction can be skipped or performed either at each hop or end-to-end, depending on their required reliability level as well as the other applications sharing the NoC. Small overhead is incurred by the fault-tolerant scheme, as effective codes are implemented (i.e. Hamming and Hsiao). Other codes can also be considered and added to the library we developed. To deal with multiple errors, block and interleaving coding can be used.

Future work

For the application checkpoint and recovery protocols, a decision algorithm that dynamically switches between blocking and non-blocking mechanisms for each checkpointing phase can be designed and implemented for the tasks that accept both approaches. Dynamically instantiation and interchangeability of the RMU configurations, depending on the communication patterns in different phases of the NoC operation, is another future direction. Evaluating our (dynamically-configured) protocols on different types of application traffic patterns (not only uniform traffic) with non-static, bursty characteristics, and long-range dependencies is another work direction to be

explored. The performance estimation of all these protocols using other application models, benchmarks or real applications is another future work direction.

As future work in 3D NoC routing, we plan to apply the routing mechanism to other topologies of 2D layers and assess its effectiveness for other types of traffic. Different instantiations of the weight function should be evaluated. VNTs in different layers must be considered to obtain a global view about the traffic load on a vertical link.

Regarding the transmission fault-tolerant schemes, future work directions are: completing the library with other codes, comparing the impact of these schemes with other approaches, and evaluating them for concrete applications.

As a longer term future work, we plan to implement a multi-level fault-tolerant NoC that integrates all the proposed fault-tolerant solutions.

Chapter 8. Résumé en français (French Summary)

Tolérance aux fautes multi-niveau dans les réseaux sur puce

Ce chapitre est un résumé en français de la thèse. Chaque section correspond à un chapitre de la version originale écrite en anglais. Après une introduction, les deux sections suivantes traitent du contexte et de la motivation pour ce travail ainsi que de l'état de l'art. Les trois sections suivantes présentent nos contributions aux différents niveaux de la pile protocolaire de communication : application, réseau, liaison de données, et transport. Les deux derniers sont traités ensemble en raison de la similitude des solutions tolérantes aux fautes respectives. La dernière section conclut la thèse et discute les travaux futurs.

8.1 Introduction

Aujourd'hui, le traitement de l'information s'éloigne de plus en plus des seuls ordinateurs personnels vers les systèmes embarqués, car les objectifs principaux sont de rendre l'information disponible à tout moment, n'importe où et de construire de l'intelligence ambiante dans notre environnement. Les systèmes embarqués ont des applications dans tous les domaines et concernent aussi bien la vie quotidienne que les missions critiques : audio, vidéo, électroménager, électronique automobile, avionique, télécommunications, systèmes médicaux, systèmes d'authentification, cryptographie, systèmes de production industrielle, robotique, etc.

Les systèmes sur puce (*systems-on-chip*, SoC) sont des systèmes embarqués qui comprennent plusieurs composants dans un seul circuit intégré (*integrated circuit*, IC) : processeurs, mémoires, périphériques, circuits de gestion d'énergie et autres. Un SoC avec plus d'un cœur de processeur est appelé système multiprocesseur sur puce (*MP-SoC*).

Le développement de la téléphonie mobile, des assistants numériques personnels (PDA), et des technologies multimédia en général crée de nouveaux besoins tels que le décodage vidéo, les jeux interactifs en trois dimensions ou le décodage audio numérique. Ces applications nécessitent de plus en plus de puissance de calcul [ITR09a].

L'intégration de plusieurs processeurs dans une seule puce est une solution potentielle pour faire face à l'exigence continue de puissance de calcul des applications embarquées et à la contrainte de temps de mise sur le marché. Le nombre des unités de calcul (*processing engines*, PE) devrait augmenter considérablement dans les années prochaines [ITR09a] et atteindre « mille cœurs » en 2020.

Face à la complexité énorme, les contraintes spécifiques des systèmes embarqués doivent être prises en compte, tels que la faible consommation d'énergie, la petite surface de silicium, la fabrication ou le rendement.

Sur le plan technologique, conformément à la *loi de Moore*, un degré plus haut d'intégration est prévu [Int05]. Bien que les performances des transistors continuent de s'améliorer avec la réduction de leurs tailles, le câblage les reliant présente des délais croissants [ITR05]. Des répéteurs peuvent être ajoutés, afin d'atténuer le délai, mais ils consomment de l'énergie et prennent de la surface. Aujourd'hui, plus de 50% de la consommation de puissance dynamique est due à des interconnexions, et ce taux devrait augmenter [MKWS04] [JLV05]. Par conséquent, l'atteinte des objectifs de puissance devient aussi important que l'atteinte des objectifs de performance [KNM04], sinon plus.

L'intégration 3D, intensément étudié ces dernières années, offre *« plus de perspective Moore »* pour les circuits intégrés, en particulier pour les systèmes sur puce dédiés au multimédia et aux applications mobiles. L'intégration 3D consiste à empiler des circuits intégrés et à les connecter verticalement, en utilisant des vias traversant le silicium (*Through-Silicon-Via*, TSV). Les interconnexions horizontales longues sont remplacées par des interconnexions verticales plus courtes, en réduisant ainsi les délais de fils globaux et la consommation globale d'énergie dynamique. Un autre grand avantage de l'intégration 3D par couches superposées est que, comparée à l'intégration homogène, l'intégration de dispositifs et de technologies hétérogènes devient possible.

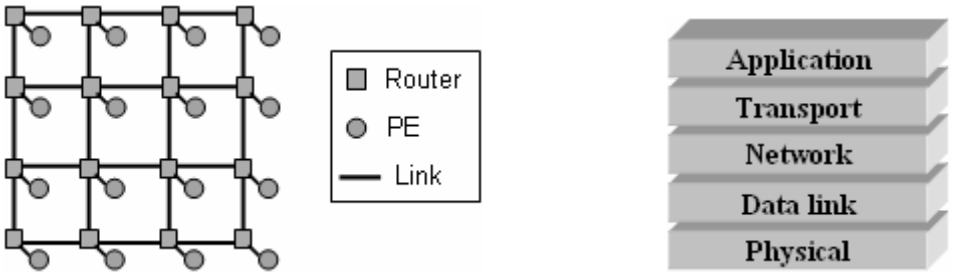
Cependant, le manque d'outils de conception traitant le *floorplanning*, le placement-routage, mais aussi les TSVs et l'empilement 3D de couches retarde actuellement l'intégration de cette technologie dans le flux principal. Outre les restrictions liées aux

irrégularités de l'hétérogénéité, des restrictions relatives au nombre, à la taille et à la position des TSV affectent la connectivité verticale de la pile [DM09]. En outre, l'intégration 3D apporte des problèmes de rendement importants, principalement en raison de processus supplémentaires de fabrication tels que l'amincissement de *wafers* et le collage.

Avec l'augmentation continue de la complexité des MP-SoC, de plus en plus d'éléments sont intégrés. Ainsi, la communication entre ces éléments devient un goulot d'étranglement pour la mise à l'échelle et les délais de mise sur le marché.

Réseaux sur puce

Les réseaux sur puce (*Network-on-Chip*, NoC) résolvent les problèmes de la complexité toujours croissante, des délais et de la dissipation de puissance des interconnexions 2D globales sur puce, en offrant un système structuré d'interconnexions constitué de *segments de fil* (*liens*) et des blocs de routage (*routeurs* ou *commutateurs*), comme représenté sur la Fig. 8-1.a. Les systèmes de base traditionnels de communication sont les connexions *point-à-point* (*P2P*) et les bus. L'architecture de communication NoC est destinée à cumuler les avantages de ces architectures principales existantes (scalabilité et latence de communication faible), tout en réduisant leurs inconvénients (tels que le nombre de longs fils de P2P et la latence de communication dans les bus).



a. Réseau sur puce b. Pile protocolaire de communication

Fig. 8-1 Réseau sur puce (a) et Pile protocolaire de communication (b)

Les applications qui utilisent un NoC sont des ensembles de tâches s'exécutant sur les différents unités de calcul (PEs) et communiquant à travers le NoC. La pile protocolaire de communication est similaire à celle proposée par ISO-OSI [Zim88] et présentée dans Fig. 8-1.b.

Pour les applications typiques SoC, la nature du modèle de communication est le plus souvent hétérogène, avec certains groupes de tâches échangeant de données plus fréquemment que d'autres. En outre, certaines tâches peuvent être critiques ou temps réel. Différentes applications avec leurs propres ensembles de tâches disjointes peuvent également s'exécuter simultanément sur le système à réseau sur puce [HCG07]. Même si les tâches dans différents ensembles disjoints n'échangent pas de données avec les autres, elles sont en concurrence pour les ressources du NoC (routeurs, liens).

Les NoCs ont tout d'abord été proposés pour les puces 2D, mais avec l'émergence de l'intégration 3D, les NoCs peuvent être étendus aux topologies 3D, compte tenu de leur modularité et scalabilité.

Le rendement et la fiabilité

Avec la mise à l'échelle de la technologie, les systèmes sur puce sont plus sensibles à différents facteurs qui ont un impact considérable sur leur rendement et fiabilité. Des défauts catastrophiques peuvent se produire dans les systèmes, en raison de défauts de

fabrication. Par ailleurs, l'impact de la variabilité est plus fort pour les éléments plus petits [ITR09b]. Ainsi, des défauts paramétriques liés à la variabilité des dispositifs et des interconnexions présentent des comportements défectueux similaires à ceux induits par des défauts catastrophiques. Les sources de ces défaillances sont : les variations de processus, les variations pendant de durée de vie et le bruit intrinsèque.

Avec la diminution des dimensions, la taille de la mémoire intégrée augmente. En même temps, le taux d'erreurs soft (*soft-error rate*, *SER*) a un impact sur la fiabilité des produits, non seulement pour les mémoires embarquées, mais aussi pour la logique et les verrous [ITR09c].

Des fautes non détectées ou non traitées et des erreurs produites au niveau physique peuvent se propager aux niveaux supérieurs de la pile protocolaire de communication et peuvent influer sur le bon fonctionnement des différents composants du système (routeurs et liens, pour les NoCs), ou peuvent conduire à la défaillance de l'ensemble du système.

Les modèles étant de taille trop importante pour un test fonctionnel d'un bon rapport coût-efficacité après la fabrication, il est nécessaire d'ajouter des mécanismes de tolérance aux pannes durant la conception, en particulier pour les applications très fiables ou critiques, comme dans les secteurs de l'automobile et l'avionique, mais pas seulement.

***

Les spécificités de l'application ainsi que les restrictions qui s'appliquent aux systèmes sur puce (tels que la consommation d'énergie, la surface et le poids) doivent être prises en compte lors de la conception des mécanismes de tolérance aux fautes. Dans cette thèse, nous nous intéressons à des solutions de tolérance aux fautes pour les NoCs, à différents niveaux de la pile protocolaire de communication.

Le reste de ce résumé est organisé comme suit (chaque section correspond à un chapitre de la thèse) :

La section 8.2 présente le contexte et la motivation pour ce travail et définit plusieurs classes de problèmes qui sont abordés dans cette thèse. Par conséquent, nous abordons les différents niveaux de la pile protocolaire de communication : l'application, le transport, le routage, et le niveau de liaison de données. Plusieurs questions ouvertes sont formulées, qui seront traitées dans les sections suivantes.

La section 8.3 analyse les travaux connexes. Nous concluons qu'il n'y a pas de solutions existantes traitant bien toutes ces questions concernant la tolérance aux fautes de NoC 2D et 3D.

La section 8.4 propose un modèle pour évaluer l'impact du recouvrement par points de contrôle (ou de reprise) sur la performance des NoCs. Des méthodes pour améliorer la scalabilité des points de reprise sont également proposées, compte tenu des spécificités de l'application.

La section 8.5 propose une méthodologie de routage inter-couche reconfigurable pour les NoCs 3D avec n'importe quelle topologie de couches 2D et de nombreuses irrégularités dans la distribution des liens verticaux. Des algorithmes de routage efficaces spécialement conçus pour les couches 2D sont réutilisés. L'adaptabilité aux spécificités de l'application est également examinée.

La section 8.6 propose un mécanisme de tolérance aux fautes pour les niveaux liaison de données et transport, adaptable aux caractéristiques de l'application. Ce mécanisme peut être appliqué à la fois sur des liens 2D et des liens verticaux dans les NoCs 3D.

La section 8.7 présente les conclusions de cette thèse et propose des perspectives.

8.2 Sûreté de fonctionnement de systèmes à base de NoC

Dans le chapitre de la thèse correspondant à cette section sont présentés les différents aspects et problèmes liés à la sûreté de fonctionnement des réseaux sur puce. La première partie présente les caractéristiques et les exigences des systèmes sur puce : la complexité, l'efficacité et la fiabilité. Les NoCs sont détaillées dans la deuxième partie : les généralités, les avantages et les inconvénients, les éléments du système basé sur NoC, la pile protocolaire de communication, la topologie des NoC et la qualité de service (QoS : *Quality of Service*). L'intégration 3D est également présentée, avec ses avantages et inconvénients, ainsi que les NoCs 3D.

Le NoC est susceptible de devenir une alternative intéressante pour la mise en œuvre des systèmes sur puce pour de nombreux domaines d'application. Certaines applications (temps réel et multimédias telles que audio / vidéo) nécessitent des bornes déterministes strictes sur les délais et les débits. Dans d'autres applications, le trafic de communication n'est pas soumis aux restrictions de délai ou de débit. Ces deux types de trafic sont appelés trafic *à débit garanti* (*guaranteed throughput*, GT) et trafic *à meilleur effort* (*best effort*, BE).

Un système contenant plusieurs applications s'exécutant en même temps peut avoir des trafics GT et BE concurrents dans le réseau. Dans de tels systèmes mixtes, la QoS du trafic BE tend à devenir faible. A côté des demandes de trafic, certaines applications / tâches peuvent être *critiques* (par exemple pour l'automobile et les applications de sécurité) et nécessitent un degré plus élevé de *fiabilité*. Dans certaines applications GT (comme le multimédia), la fiabilité peut être négociée pour le débit. Il y a aussi des cas où les deux conditions sont importantes (GT et une grande fiabilité / intégrité des données).

Considérant les différentes exigences de qualité de service pour diverses applications ou tâches d'une même application, la question suivante apparaît :

Question : *Comment faire pour atteindre des niveaux de fiabilité acceptable avec un impact réduit sur les exigences de qualité de service tout en offrant de l'adaptabilité aux spécificités des applications (intégrité des données, garantie de débit, meilleur effort) ?*

Cette question est générale et, par conséquent, doit être prise en compte à chaque mécanisme de tolérance aux fautes qui est mis en œuvre dans le système.

Une partie de ce chapitre est consacrée à la notion de sûreté de fonctionnement, avec ses attributs, menaces et moyens. Les attributs de la sûreté de fonctionnement sont les suivants : la disponibilité, la fiabilité, la maintenabilité, la sûreté et la sécurité. Les fautes, les erreurs et les défaillances représentent les menaces de la sûreté de fonctionnement. La relation entre les menaces de fonctionnement est connue comme la *chaîne faute-erreur-défaillance* [ALRV00], représenté dans la Fig. 8-2. Le mécanisme de base peut se résumer comme ceci : une faute peut entraîner une erreur qui, à son tour, peut conduire à une défaillance.

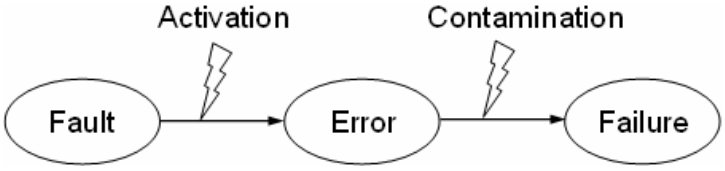


Fig. 8-2 La chaîne faute-erreur-défaillance

Etant donné que la sortie des données d'un service peut être l'entrée d'un autre, une défaillance peut se propager d'un service à l'autre service comme une faute ; ainsi une *chaîne* peut être formée de la forme : faute –> erreur –> défaillance –> erreur etc. La chaîne faute-erreur-défaillance peut être considérée à différents niveaux hiérarchiques (système complet, équipement, composant, bloc, porte logique, transistor etc.) et s'applique à la fois au matériel et au logiciel. Les menaces de la sûreté de fonctionnement affectent également les différents niveaux de la pile protocolaire de communication et peuvent également se propager entre ceux-ci.

Deux mesures typiques de la sensibilité aux défaillances sont les suivants : *le taux de défaillance*, indiquant le nombre de défaillances prévu dans le circuit dans un intervalle de temps et le *temps moyen entre défaillances* (*mean time between failures*, MTBF), qui indique le temps moyen entre deux défaillances successives. Parfois le MTBF est remplacé par sa variante, le *temps moyen avant défaillance* (*mean time to failure*, MTTF), habituellement utilisé pour les composants qui peuvent être remplacés après la première défaillance.

Les moyens de sûreté de fonctionnement sont destinés à réduire le nombre global de défaillances d'un système. Considérant le mécanisme de la chaîne faute-erreur-défaillance, il est possible de proposer des moyens pour briser ces chaînes et améliorer ainsi la sûreté de fonctionnement du système. Quatre moyens sont identifiés : *la prévention de fautes*, *la prévision de fautes*, *la suppression de fautes* et *la tolérance aux fautes*. La dernière consiste à ajouter des mécanismes dans le système permettant la prestation des services requis, même en présence de fautes (parfois à une performance dégradée). Du fait que les fautes soient diversifiées par leur type, nature, taux et lieu d'apparition, leurs effets sont difficiles à prédire et à prévenir. Ainsi, dans les systèmes complexes à base de NoC, la tolérance aux fautes est une nécessité.

En ce qui concerne la pile protocolaire de communication, différentes solutions de tolérance aux fautes peuvent être appliquées à chaque niveau : récupération par point de contrôle au niveau de l'application, routage tolérant aux fautes au niveau du réseau et récupération par correction de l'erreur ou détection de l'erreur et retransmission appliquées aux niveaux liaison de donnée et transport.

- Les techniques de récupération par points de contrôle (*Checkpoint and Rollback*, C&R) deviennent pertinentes pour les systèmes multi-processeurs sur puce, où le nombre de PEs devrait augmenter considérablement dans l'avenir proche (des centaines voire des milliers) [ITR09a]. Considérant également l'augmentation du nombre de fautes qui apparaissent avec la miniaturisation importante des systèmes intégrés, l'amélioration de la scalabilité de la récupération par points de reprise devient décisive. Dans ce contexte, la question suivante se pose :

Question : *Comment l'incidence du mécanisme de récupération par points de contrôle sur la performance de NoC peut être évaluée et comment peut être améliorée la scalabilité de ce mécanisme ?*

- Compte tenu des caractéristiques de la topologie de NoC 3D (connexion verticale incomplète et topologies peut-être différentes dans chaque couche), ainsi que les éventuelles défaillances de routeurs et de liens, la question suivante se pose :

Question : *Comment faire pour router les messages dans un NoC 3D ayant une connectivité verticale partielle et des topologies de couches potentiellement différentes, avec un effort de conception, des surcoûts et des délais de mise sur le marché minimaux, tout en assurant un routage optimal à l'intérieur de chaque couche ?*

- Tenant compte du faible rendement et qualité de TSV (*Through-Silicon-Via*), des méthodes pour améliorer la fiabilité des liens doivent être considérées avant de les

déclarer comme défaillantes. Des fils de rechange et la sérialisation sont actuellement utilisés pour traiter les fautes permanentes. Cet aspect n'est pas abordé dans cette thèse.

Question : *Comment faire face aux fautes transitoires et intermittents dans les liens de NoC ?*

Compte tenu de l'étroite collaboration entre les éléments du système, nous concluons que des mécanismes de tolérance aux fautes sont nécessaires pour tous les composants et à tous les niveaux de la pile protocolaire de communication, afin de limiter la propagation et les effets des fautes et erreurs inévitables, évitant ainsi la défaillance de l'ensemble du système. La capacité de tolérance aux fautes par rapport au compromis performance / coût doit être examinée pour chaque type d'application, en fonction de sa qualité de service spécifique. Dans ce contexte, la capacité d'adaptation et de reconfiguration sont des points clés. Suite à cette analyse, plusieurs questions ont émergé. Des réponses aux trois dernières questions sont examinées dans les chapitres / sections suivant(e)s, en tenant compte des réponses à la première.

8.3 État de l'art des techniques de tolérance aux fautes dans les NoCs

Cette section présente l'état de l'art des méthodes de tolérance aux fautes mises en œuvre dans les NoCs. La première partie concerne les techniques qui permettent la reprise de l'application après erreurs ou défaillance au moyen de points de contrôle et de restauration. Les défaillances d'éléments du NoC (routeurs et liens) sont traitées par des algorithmes de routage tolérants aux fautes, dont une vue d'ensemble est donnée dans la deuxième partie de la section. Avant de déclarer un élément du NoC comme défaillant, des techniques de tolérance aux fautes sont utilisées pour améliorer sa fiabilité. Des méthodes pour tolérer des fautes transitoires lors des transmissions sur le NoC sont présentées dans la dernière partie de la section.

8.3.1 Récupération par points de reprise

Différents protocoles de restauration de l'application par points de contrôle ont été développés pour les macro-réseaux. Les principales approches afin d'établir des points de contrôle sont : la prise coordonnée de points, la prise non-coordonnée et celle induite par la communication. Différents types de journalisation des messages (optimiste, pessimiste, de causalité) sont utilisés pour compléter les points de reprise, et le journal est fait soit par l'expéditeur ou par le destinataire. L'amélioration de la scalabilité de l'approche de prise de points de contrôle coordonnée a été abordée, en réduisant le nombre de messages de synchronisation et celui de points de contrôle. Des protocoles non-bloquants ont également été proposés, tels que la prise de points de contrôle par groupes, dans une mémoire commune. La portabilité des points de reprise sur différents plateformes a été aussi abordée. Celle-ci peut être très utile pour les SoCs hétérogènes [LMS+05]. Cependant, les protocoles proposés traitent de manière égale toutes les tâches de l'application. Dans les SoC, les motifs de communication peuvent être hétérogènes et différents ensembles de tâches peuvent être exécutées sur la même puce, indépendamment ou simultanément. Par conséquent, les mécanismes de tolérance aux fautes doivent introduire une pénalité minimale afin de respecter les exigences de qualité de service.

8.3.2 Routage 3D et tolérance aux fautes

De nombreuses approches de routage 2D ont été proposées pour les systèmes distribués ainsi que pour les NoCs. Quelques algorithmes de routage existent également pour des topologies 3D de maille et tore. Tous ces algorithmes proposent des solutions très complexes, plutôt adaptées pour les macro-réseaux, où les surcoûts et les latences engagés ne sont pas aussi critiques que pour les systèmes sur puce. En outre, ils traitent tous de topologies 3D régulières de mailles ou tores, modèles qui ne sont pas facilement mis en œuvre pour les NoCs 3D réels, comme indiqué dans la section précédente. Les topologies de NoC 3D peuvent avoir une connectivité verticale incomplète et / ou des topologies différentes dans les couches 2D. Pourtant, chaque couche 2D peut avoir une topologie interne régulière. Une possibilité d'utiliser les algorithmes de routage 3D ci-dessus dans les NoCs 3D serait de modéliser les irrégularités des NoCs 3D comme des défauts de la maille / tore 3D régulière. Toutefois, compte tenu de l'irrégularité élevée de NoCs 3D, ces algorithmes sont dans ces cas susceptibles de dégrader considérablement les performances. Les algorithmes de routage pour des topologies générales sont toujours une solution possible. Cependant, la généralité impliquant des surcoûts plus élevés (surcharge et latence), les algorithmes de routage pour des topologies (partiellement) régulières sont en générale préférables.

8.3.3 Schémas de contrôle d'erreurs

Les codes détecteurs / correcteurs d'erreurs sont largement utilisés dans les différentes composantes du système (liens et routeurs, dans le cas des NoCs) pour tolérer des erreurs temporaires. Ils complètent d'autres techniques d'amélioration de la sûreté de fonctionnement du système (par exemple des fils de rechange et la sérialisation pour améliorer le rendement de fils). La récupération des erreurs transitoires dans les transmissions est réalisée soit par la correction des erreurs soit par la détection puis la retransmission de la donnée. Ces mécanismes peuvent être appliqués à chaque lien ou de bout en bout, au niveau d'un flit ou d'un paquet. Des codes ayant une haute capacité de détection / correction peuvent être mis en œuvre, mais leur coût est souvent prohibitif. La réduction de la consommation d'énergie est l'une des principales demandes qui limitent les mécanismes de tolérance aux fautes, étant une des caractéristiques générales des systèmes sur puce. La latence est critique, notamment pour les services de débit garanti. Des solutions hybrides et adaptatives ont été proposées, tenant compte de différents niveaux de tolérance aux fautes pour des exigences spécifiques de qualité de service de l'application. Toutefois, celles-ci proposent l'utilisation de différents types de codes et de différents mécanismes de recouvrement pour chaque cas. La commutation dynamique entre eux implique des surcoûts élevés, en raison de la présence de tous ces mécanismes dans le dispositif.

8.4 Récupération de l'application par points de reprise dans les NoCs

Cette partie présente des protocoles de tolérance aux fautes au niveau de l'application, basés sur le concept de récupération. Nous évaluons l'efficacité et l'impact sur les performances de la reprise par points de contrôle dans les NoCs et nous proposons des améliorations de leur scalabilité et afin de faire face aux différentes exigences de qualité de service. Les principales classes d'algorithmes de récupération sont basées sur la

coordination et la non-coordination des points de reprise. Différentes solutions visant à améliorer les performances de ces algorithmes, en exploitant d'une part le partitionnement de l'application et d'autre part la fonctionnalité de blocage durant la prise de point de contrôle sont discutées. Des solutions pour l'amélioration de la scalabilité au niveau protocole ainsi qu'au niveau routage sont également proposées.

En disposant d'une bibliothèque de solutions, la plus appropriée en termes de performances / surcoûts peut être sélectionnée pour chaque application, en fonction de différents facteurs, comme les caractéristiques de l'application, la qualité de service et le taux de défaillance prévu.

8.4.1 Le principe de fonctionnement

Le mécanisme de récupération de l'application par points de reprise et restauration (*checkpoint and rollback*, C&R) permet au système de reprendre l'exécution après une défaillance, à partir d'un *état cohérent* (ou *point de reprise global* ou *ligne de récupération*) antérieur à la défaillance. Le concept de récupération par points de reprise par rapport un redémarrage complet est illustré dans la Fig. 8-3.

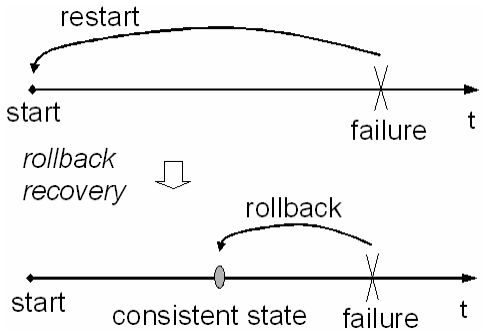


Fig. 8-3 Récupération vs. redémarrage

Dans le contexte des NoCs, la technique de récupération par points de contrôle est destinée à traiter les fautes *transitoires* ou *intermittentes* (voir chapitre 2) qui peuvent provoquer des défaillances de l'application, ainsi que des *défaillances* permanentes des IPs. Dans le premier cas, l'exécution de l'application peut être reprise sur le même IP, alors que dans le second, elle doit être reprise sur un autre IP. Si une sérialisation des points de contrôle appropriée est utilisée, l'IP défaillant et le nouveau IP peuvent avoir des architectures complètement différentes [ZP06] [DM05] [RS97].

Le mécanisme de prise de points de reprise (ou points de contrôle) signifie sauvegarder périodiquement sur un *stockage stable*, lors de l'exécution sans erreur, des informations nécessaires et suffisantes pour reconstruire un *état cohérent* en cas de défaillance. Le *stockage stable* est un élément de mémoire dont le contenu persiste et n'est pas corrompu lors les défaillances tolérées. Un état de l'application est cohérent s'il pouvait être atteint suite à une exécution correcte et sans erreur de l'application du point de départ.

Avant d'enregistrer tout état, l'application doit passer un *test d'acceptation* [Par96] [ASK08b], pour éviter l'enregistrement d'un état déjà contaminé par des erreurs. La rédaction du test d'acceptation est la partie la plus critique du mécanisme de point de contrôle et de restauration, et nécessite une bonne connaissance de l'application

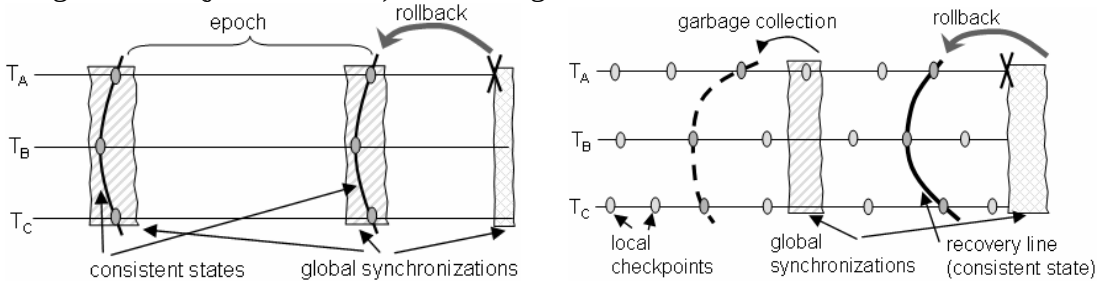
[ASK08a]. En raison de sa forte dépendance à chaque application, le test d'acceptation n'est pas détaillé dans ce travail.

8.4.2 Prise coordonnée et non-coordonnée de points de contrôle

Différentes approches de récupération par points de contrôle ont été proposées. La différence entre elles consiste en **quand** et **comment** l'état cohérent est déterminé. Les principales approches sont la prise de points de contrôle coordonnée et celle non coordonnées.

Dans l'approche coordonnée de prise de points de reprise, *coordinated checkpointing* CC, (voir Fig. 8-4.a), les tâches se synchronisent pendant l'exécution normale (sans défaillance) et enregistrent un état cohérent sur le stockage stable. En cas de défaillance, ils reprennent l'exécution du dernier état cohérent. Cette approche est préférable dans la pratique, compte tenu de sa simplicité et son faible surcoût.

Dans l'approche non coordonnée de prise de points de reprise, *uncoordinated checkpointing* UC, (voir Fig. 8-4.b), les tâches prennent des points de contrôle lors de l'exécution normale sans aucune coordination. En cas de défaillance, ils se synchronisent afin d'établir l'état cohérent le plus récent à partir de leurs points de contrôle individuels, puis ils récupèrent de cet état. L'avantage principal est qu'aucune synchronisation n'est nécessaire pour prendre les points de contrôle, mais l'approche est soumise à l'effet domino [EAWJ02] en cas de défaillance. L'effet domino se produit lorsque, à partir de la plus récente série de points de contrôle individuels des tâches, la ligne de récupération régresse pour une ou plusieurs tâches à la fois, au point de contrôle individuel antérieur. La cause de cette régression est l'incohérence de la ligne de récupération. À son tour, chaque régression peut conduire à de nouvelles régressions. Cette opération est répétée jusqu'à ce qu'un état cohérent puisse être établi. Dans le pire des cas, la première ligne de récupération cohérente qui peut être établie est identique au point de départ de l'application. Une solution pour limiter l'effet domino est de compléter l'UC avec l'enregistrement (journalisation) des messages.



a. Points de contrôle coordonnés (CC) b. Points de contrôle non coordonnés (UC)

Fig. 8-4 Points de contrôle coordonnés (a) et points de contrôle non coordonnés (b)

Il y a plusieurs caractéristiques communes aux protocoles coordonnés et non coordonnée que nous avons développés et adaptés pour les NoCs.

Lors des phases de prise de points de contrôle et de restauration, un coordonateur unique, appelée unité de gestion de récupération ou RMU (*recovery management unit*) est utilisé pour les synchronisations, aussi dans le cas de CC et de UC. Toutefois, toutes les tâches peuvent initier des phases de synchronisations globales, en envoyant un message à la RMU. La RMU unique évite les phases d'élections de coordonateur et autres synchronisations connexes.

Une autre similitude est que les défaillances multiples sont tolérées par les deux approches. Les défaillances peuvent survenir n'importe quand, même dans les phases de synchronisation. La RMU est également protégée contre les défaillances à l'aide d'un stockage stable pour les informations critiques.

Une autre similarité est que les messages sont enregistrés dans les deux cas. Cependant, dans le cas de points de contrôle non coordonnés, tous les messages sont enregistrés, mais pas tous seront utilisés pour une éventuelle récupération de l'application. Dans le cas des points de contrôle coordonné, seuls les messages nécessaires pour la récupération sont enregistrés, vu qu'ils peuvent être identifiés au moment de l'enregistrement.

Le nombre de points de contrôle nécessaires dans le cas non coordonné est au moins de deux : l'un appartenant à la ligne de récupération la plus récente et le nouveau qui est en train d'être pris. Plusieurs points de contrôle peuvent être pris jusqu'à ce qu'une ligne de récupération puisse être établie et que les points de contrôle anciens puissent être supprimés. Avoir une ligne de récupération très récente est important, car l'objectif est de réduire au minimum la quantité de traitement à refaire après une défaillance. Cependant, avoir une ligne de récupération récente implique une prise de points de contrôle fréquente. D'autre part, la prise de points de contrôle implique un surcoût à l'exécution normale de l'application. De plus, avoir beaucoup de points de contrôle a pour conséquence de fréquentes phases de récupération en mémoire (*garbage collection*), ce qui induit également de la latence. Compte tenu de ces faits, la fréquence des points de contrôle doit avoir une limite supérieure. À l'autre extrême, si la fréquence des points de contrôle est trop faible par rapport à la fréquence de défaillance, la ligne de récupération ne peut pas avancer entre deux défaillances successives. Par conséquent, la fréquence de point de contrôle doit être établie en fonction de la fréquence de défaillance.

Dans le cas d'une prise coordonnée de points de contrôle, au plus deux points de contrôle sont nécessaires, celui qui est déjà validé et le nouveau. Une fois que le nouveau est validé, le précédent peut être supprimé.

Les simulations montrent que pour des faibles taux de défaillance, la méthode coordonnée de prise de points de contrôle est plus efficace que celle sans coordination. Elle est également préférable du fait de son surcoût réduit, en particulier pour des charges de trafic élevées. Toutefois, si le taux de défaillance augmente, la méthode coordonnée devient inefficace. Dans ces cas, la méthode non coordonnée combinée avec la journalisation des messages devient pertinente, en dépit du surcoût beaucoup plus élevés. En outre, le point de croisement de l'efficacité des deux méthodes de récupération se produit plus tôt si la charge de trafic augmente.

8.4.3 Adaptabilité à la QoS et amélioration de la scalabilité

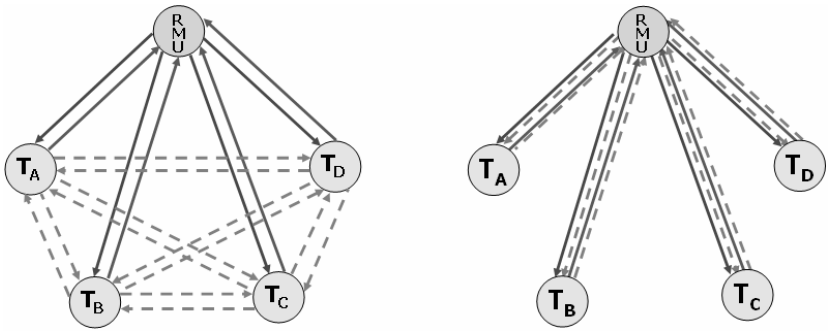
8.4.3.1 Prise coordonnée de points de contrôle avec réduction du nombre des diffusions

Compte tenu de sa simplicité dans la synchronisation d'un état global, cohérent et sans faute, la prise coordonnée de points de contrôle est préférable dans la pratique. Cependant, avec le nombre croissant de PEs, la méthode coordonnée souffre d'une mauvaise scalabilité [EP04]. Ceci est dû au fait que la synchronisation globale nécessaire pour construire un état global cohérent induit un surcoût de communication important pour les grands systèmes, ce qui augmente la latence des points de contrôle (le temps nécessaire à

l'exécution du protocole pour enregistrer un nouveau point contrôle, y compris la synchronisation globale). La période minimale de points de contrôle est limitée par la latence de points de contrôle. Par conséquent, une grande latence de points de contrôle force des périodes de points de contrôle plus larges. En cas de défaillance, ces périodes plus larges donnent lieu à des plus ample restaurations, et par conséquent à des temps de récupération plus importants (puisque l'intervalle d'exécution depuis le point de contrôle est perdu), impactant ainsi la performance du système. En outre, si la durée de synchronisation est plus grande que le MTTF, il n'y a pas de temps pour prendre de nouveaux points de contrôle global entre deux défaillances successives. Dans de tels cas, des récupérations sont effectuées au même ancien point de contrôle et donc, l'exécution de l'application n'avance pas. Dans le cas d'une coordination de la prise de points de contrôle, la durée de synchronisation globale est aussi appelée durée de point de contrôle (global). Par conséquent, dans le cas d'une coordination de la prise de points de contrôle, non seulement la surcharge induite par la prise des points de contrôle local doit être réduite, comme dans le cas de la prise non coordonnées de points de contrôle, mais aussi la durée de synchronisation globale, puisque les points de contrôle locaux n'ont pas de sens tous seuls dans ce cas (c.-à-d. CC).

Notre objectif est d'améliorer la scalabilité du mécanisme de récupération par points de contrôle en réduisant la durée de synchronisation globale et donc la latence de point de contrôle.

Lorsque le protocole classique de prise de points de contrôle coordonnée est utilisé, plusieurs diffusions (*broadcasts*) sont effectuées. Les diffusions qui chargent de façon significative la structure de communication sont celles qui sont envoyées entre les tâches pour s'informer mutuellement qu'elles ont commencé leurs phases de prise de point de contrôle. Ces messages sont appelés CK_START et sont marqués par des lignes pointillées sur la Fig. 8-5.a. Ils introduisent une surcharge d'environ n^2 messages dans le réseau, où n représente le nombre de PEs dans le système. Ce nombre devient important lorsque le nombre de PEs augmente : 12 messages pour un NoC en maille 4x4, mais 65280 messages pour un NoC en maille 16x16! En outre, l'envoi de ces messages est un processus en rafale, et aura tendance à créer de la congestion, et par conséquent, à augmenter la latence de points de reprise.



a. Protocole coordonné classique b. Protocole avec nombre réduit de diffusions

Fig. 8-5 Réduire le nombre de diffusions dans le protocole coordonné

Dans le protocole coordonné optimisé, les diffusions « tous-à-tous » entre les tâches sont remplacées par la transmission d'un seul message par tâche (CK_START envoyé à la RMU par toutes les tâches) et une diffusion (CK_START_ALL envoyée à toutes les tâches par la RMU). Ces messages sont marqués par une ligne pointillée sur la Fig. 8-5.b.

Le nombre de messages de synchronisation est donc réduit de $O(n^2)$ à $O(n)$. Le protocole de restauration associé est inchangé.

Les résultats de simulation montrent une amélioration significative de la scalabilité lorsque la prise de points de contrôle utilise le protocole avec un nombre réduit de diffusions.

8.4.3.2 Prise de points de contrôle coordonnée bloquante et non bloquante

La prise de points de contrôle coordonnée avec blocage (B) et coordonnée sans blocage (NB) ont été toutes deux proposées (les tâches bloquent ou non leur exécution normale au cours de la prise de points de contrôle).

Les durées de prise de points de contrôle avec et sans blocage sont approximativement les mêmes pour des charges très faibles de trafic. Toutefois, avec l'augmentation de la charge de trafic, cette durée augmente de manière significative dans le cas non-bloquant, alors que dans le cas bloquant, celle-ci n'est pas affectée. Ceci s'explique par l'existence ou non du trafic « supplémentaire » dans le NoC, dû à l'application. Dans le cas non-bloquant, le trafic de récupération est considérablement retardé par le trafic des applications. Dans le cas avec blocage, comme le trafic de l'application est arrêté, le trafic de récupération n'est pas retardé.

Normalement, l'exécution globale de l'application est retardée dans l'approche avec blocage, puisque l'application est bloquée durant les phases de prise de points de contrôle. Cela peut être observé pour des taux de défaillance relativement faibles. Toutefois, pour des taux de défaillance plus élevés, l'approche avec blocage induit une latence inférieure à celle de l'approche sans blocage. En outre, l'approche non-bloquante devient inefficace pour des taux de défaillance plus élevés, ce qui n'est pas le cas pour celle avec blocage. En fait, comme la durée de prise de points de reprise est plus grande dans le cas sans blocage que dans celui avec le blocage, l'intervalle de temps entre deux défaillances successives n'est pas assez long pour pouvoir prendre un nouveau point de contrôle sans blocage. Ainsi, dans l'approche non-bloquante, la probabilité de restauration à partir du même ancien point de contrôle augmente. En conséquence, des parties de l'application sont ré-exécutées à plusieurs reprises, ce qui conduit à une latence extrêmement élevée. Pour des charges de trafic supérieures, la même tendance est maintenue, mais le point d'intersection entre les deux approches se produit pour une valeur inférieure du taux de défaillance.

Ainsi, des approches différentes (avec blocage ou sans blocage) sont préférables pour des charges de trafic et taux de défaillance différents.

Nos deux protocoles coordonnés (le protocole de base et celui avec le nombre réduit de diffusions) peuvent être réalisés avec ou sans blocage de l'exécution de la tâche durant la prise de point de contrôle. La même séquence d'actions est exécutée dans les deux approches, si ce n'est que l'exécution de la tâche est bloquée pendant la prise bloquante. Toutefois, pendant la prise de point de contrôle, les messages tardifs (*late*) sont enregistrés et feront partie du point de contrôle. Ce choix (au lieu d'attendre que les canaux du réseau soient vides) réduit la durée de prise de points de contrôle. Puisque le protocole avec blocage et celui sans blocage sont les mêmes, chaque tâche peut participer à la prise globale de points de contrôle, en bloquant ou non son exécution normale, indépendamment des autres tâches. Un état global cohérent est sauvegardé, indépendamment de la configuration (bloquante / non-bloquante) des tâches.

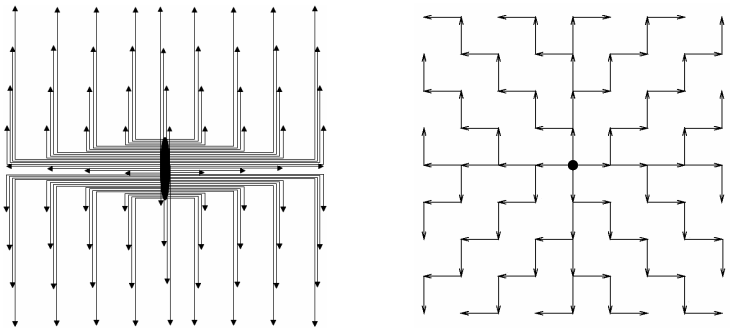
La décision de bloquer ou non l'exécution normale peut être prise pour chaque tâche et pour chaque point de contrôle global, en fonction des exigences de qualité de service de l'application et du trafic dans le NoC. Par exemple, si l'exécution d'une tâche est critique à l'application ou en temps réel et ne doit pas être interrompue pendant la prise du point de contrôle global, celle-ci participera à la prise de point de contrôle sans bloquer son exécution. D'autre part, si une tâche entraîne une charge de trafic élevé, mais que son exécution n'est pas critique, elle sera bloquée. Cela est dû au fait que la durée de prise de points de contrôle peut être étirée exactement par le trafic élevé dans le NoC ; ainsi, bloquer les tâches qui induisent un tel trafic peut contribuer à la réduction de cette période.

Dans notre modèle, la décision B-NB d'une tâche est indépendante des choix des autres tâches, car les tâches peuvent enregistrer leur état à tout moment. Sans cette possibilité, d'autres restrictions sur le choix de B-NB doivent être considérées, comme la décision commune pour les groupes de tâches communicantes.

8.4.3.3 Diffusion optimisée

Dans la sous-section 8.4.3.1, nous avons montré que lorsque le nombre de diffusions (*broadcasts*) est réduit dans le protocole de synchronisation CC, la scalabilité en est améliorée. Cependant, une seule diffusion peut provoquer de la congestion dans le réseau, si un routage classique point-à-point est utilisé. Dans cette sous-section nous proposons l'utilisation d'une diffusion optimisée pour la synchronisation globale des tâches. Nous considérons des topologies en maille (*mesh*) pour l'illustrer, mais le même raisonnement peut être appliqué pour d'autres topologies.

Un algorithme classique de diffusion à *n* nœuds injecte *n* messages dans le réseau. Comme il n'y a pas de lien dédié entre tous les PEs, les messages envoyés par une diffusion suivent certains liens communs avant d'arriver à leurs destinations respectives. Ainsi, un nœud près de la source de diffusion doit passer plusieurs messages à d'autres nœuds, alors que leur contenu est identique. La Fig. 8-6.a illustre le chargement des liens dans un scénario classique de diffusion à partir du nœud central d'un maillage 9x9, en utilisant un routage statique XY. Les liens horizontaux sur la même ligne que le nœud expéditeur doivent supporter une charge de trafic très élevée, en particulier les liens plus proches de l'expéditeur. Il peut être observé dans la figure que la charge sur les liens est très déséquilibrée.



a. Diffusion classique par routage XY b. Diffusion optimisée

Fig. 8-6 Diffusion classique vs. diffusion optimisée

Un exemple de diffusion efficace « un-à-tous » pour les réseaux maillés est illustré dans la Fig. 8-6.b, pour le même maillage 9x9. Cette diffusion est basée sur un algorithme simple [YW99] qui implique la duplication de message à la volée. Le nœud

source envoie le message à diffuser uniquement à ses voisins de premier rang (distance Manhattan de 1). Ensuite, chaque nœud fournit le message à son PE associé et transmet des copies de celui-ci à ses voisins de premier rang, de façon que chaque nœud reçoive le message une seule fois. La charge de trafic sur les liens en est alors équilibrée et minimale.

Les résultats obtenus montrent l'efficacité de cette technique appliquée à la prise coordonnée de points de contrôle par rapport à une diffusion XY classique. Les résultats de simulation indiquent que l'amélioration relative, en termes de latence et de surcoût mémoire, devient plus importante avec l'augmentation du nombre de PEs, en améliorant significativement la scalabilité du mécanisme de récupération par points de contrôle.

8.4.3.4 Configuration des partitions

Les applications fonctionnant sur les plates-formes MPSoC peuvent être partitionnées en plusieurs groupes de communication. En outre, plusieurs applications distinctes peuvent fonctionner en même temps dans un MPSoC. Ainsi, on peut considérer séparément les différentes partitions de l'application(s) pour les points de reprise. Ainsi, les caractéristiques essentielles des protocoles de tolérance aux fautes peuvent être améliorées, c'est-à-dire que le temps pour déterminer la ligne de récupération (dans le cas de UC) et la durée de synchronisation globale (dans le cas de CC) peuvent être réduits. Nous proposons deux configurations de récupération optimisées pour le protocole coordonné, qui peuvent également être adaptées et appliquées à l'approche non coordonnée de prise de points de contrôle.

Les configurations de récupération optimisées profitent des phases d'opération de NoC lorsque le modèle de communication est partitionné. Ainsi, afin de réduire la latence de point de contrôle, il suffit que chaque tâche prenne son point de contrôle en coordination seulement avec les tâches de sa propre partition. Dans la Fig. 8-7, les deux configurations optimisées de RMU (unité de gestion de récupération) sont représentées : *RMU partagée* et *RMU multiples*. Les nœuds exécutant des tâches dans la même partition sont représentés avec la même texture (dans cet exemple nous supposons que si plusieurs tâches sont en cours d'exécution sur une même unité de calcul, ils appartiennent à la même partition de l'application).

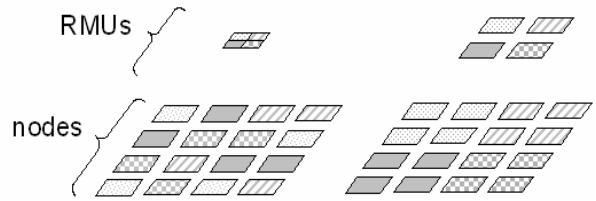


Fig. 8-7 Configurations optimisées de RMU

Pour le cas RMU partagée (Fig. 8-7.a), la même RMU est utilisée par toutes les partitions pour toutes leurs synchronisations intra-partition. Une RMU partagée est plus complexe, car elle doit être capable de gérer les synchronisations de plusieurs partitions. Symboliquement, elle est représentée de plusieurs parties gérant indépendamment chaque partition. Toutefois, les synchronisations de partition ne sont pas traitées séquentiellement (c'est-à-dire une nouvelle synchronisation d'une partition ne doit pas attendre la fin de l'actuelle), ainsi cette RMU réduit ces latences en permettant le recouvrement des synchronisations des partitions différentes.

Dans la configuration RMU multiples (Fig. 8-7.b), chaque partition a une RMU dédiée pour ses synchronisations intra-partition. Chaque RMU est représentée avec la même texture que la partition correspondante. Dans cette configuration, les RMUs ont besoin de gérer les synchronisations dans une seule partition ; ils agissent comme des RMUs simples pour leur partition respective. Cette solution nécessite des coûts plus élevés, vu que plusieurs RMUs doivent être disponibles dans le NoC. Toutefois, elle permet de meilleures performances lorsque les tâches de communication sont localisées, car chaque RMU peut être située à proximité de sa partition, ou, idéalement, dans le point central de la partition. Ainsi, la latence de communication entre les tâches et la RMU correspondante est réduite.

Afin de suggérer l'avantage de la localisation de RMUs multiples, les tâches d'une même partition sont représentées comme localisées dans la Fig. 8-7.b et non localisées dans la Fig. 8-7.a. Toutefois, les deux configurations peuvent être utilisées que les partitions soient localisées ou non.

Les résultats de simulation montrent que les configurations de RMU partagée et multiples améliorent considérablement les capacités de récupération du système pour des taux de défaillance élevés. Pour un maillage de 4x4, les performances entre RMU partagée et RMU multiples sont comparables, mais avec un coût plus réduit pour le cas RMU partagée. De plus, il est à noter que d'autres configurations peuvent également être utilisés, tels que plusieurs RMUs partagées.

8.5 Routage inter-couche tolérant aux fautes reconfigurable pour les NoCs 3D

Cette partie présente un mécanisme de routage inter-couche reconfigurable (appelé RILM) pour le NoCs 3D. Comme exposé dans le deuxième chapitre de la thèse, les NoCs 3D avec une connectivité verticale partielle sont préférés et chaque topologie de couches peut être différente des autres. Ces irrégularités des NoCs 3D sont prises en compte par RILM, tout en utilisant des algorithmes de routage bien optimisés dans chaque couche 2D de la pile. On considère que les couches 2D de la pile 3D ne sont pas partitionnées. Nous prouvons que RILM est tolérant aux fautes. En outre, si tous les algorithmes de routage dans chaque couche 2D sont tolérants aux fautes, la tolérance aux fautes du routage est obtenue pour d'ensemble de la pile des couches.

L'algorithme de routage 3D composé à l'aide RILM est présenté dans la première sous-section. Les sous-sections suivantes exposent la configuration initiale et la reconfiguration du RILM, tant pour le cas particulier des topologies de graphes non-orientés que pour le cas général des topologies de graphes mixtes. RILM construit des arbres enracinés dans des nœuds verticaux (*vertical node tree*, VNT), qui sont utilisés pour les reconfigurations.

8.5.1 Algorithme de routage 3D

L'idée de base de l'algorithme de routage 3D à l'aide de RILM est de passer le message à la couche destination dans un premier temps (s'il n'est déjà pas là), puis de le router dans cette couche, jusqu'au nœud destination. Pour atteindre la couche de destination, le message doit passer par d'autres couches, dites couches *intermédiaires*. Pour quitter une couche intermédiaire pour la prochaine, le message peut avoir besoin de se déplacer d'abord dans la couche courante afin de parvenir à un nœud vertical qui a un port de

sortie vers la couche suivante. Pour les itinéraires intra-couche, l'algorithme de routage de la couche 2D courante est utilisé.

L'algorithme global de routage 3D est présenté dans l'algorithme Algorithm 8-1. On note la source et la destination d'un message par *S* et *D*, respectivement, et le nœud actuel par *C*. Par *z*, on note la dimension verticale. L'indice de la couche d'un nœud *N* est noté par le z_N. Nous désignons par V_{To}/V_{From} un nœud vertical (ou 3D) qui a des ports de sortie / d'entrée vers / de liens verticaux. Si la direction est spécifiée, le nœud vertical est dénoté par : V_{ToUp}, V_{ToDown}, V_{FromUp}, $V_{FromDown}$. Par exemple, dans Fig. 8-8, le nœud 1 est V_{ToDown} et le nœud 13 est à la fois V_{FromUp} et $V_{FromDown}$.

Algorithm 8-1. Routage 3D – Vue globale

```
1: tant que z_D <> z_C
2:   si z_D plus haut que z_C
3:     alors V_To = V_ToUp
4:     sinon V_To = V_ToDown
5:   fin si
6:   routage 2D vers V_To
7:   aller à la couche suivante
8: fin tant que
9: routage 2D vers D
```

Notez qu'un nœud 3D n'a pas nécessairement à la fois un lien vers la couche supérieure et un autre vers la couche inférieure. Par conséquent, deux nœuds verticaux doivent être connus par chaque nœud du NoC 3D, un pour chaque sens vertical : V_{ToUp} et V_{ToDown}. Un seul nœud vertical est suffisant pour la couche en dessus et pour celle en dessous de la pile. La manière dont V_{ToUp} et V_{ToDown} sont déterminés est détaillée dans la sous-section suivante.

Dans la Fig. 8-8, un exemple de routage 3D en utilisant l'algorithme présenté ci-dessus est représenté. Supposons que la source du message est le nœud 34, dans la couche L_2, et la destination est le nœud 12, dans la couche L_0. Supposons que certains nœuds et liens sont défectueux. Les nœuds défectueux sont marqués en foncé (8, 24 et 31) et les liens défectueux sont marqués avec des x.

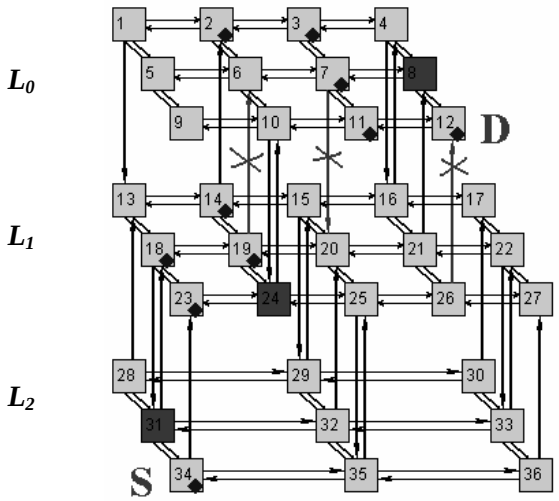


Fig. 8-8 Example de routage dans un NoC 3D

L'itinéraire est déterminé selon l'algorithme de routage proposé. Les nœuds de l'itinéraire sont marqués par des losanges (Fig. 8-8) : 34, 23, 18, 19, 14, 2, 3, 7, 11 et 12.

8.5.2 Attribution de nœuds verticaux pour le routage avec RILM

L'algorithme de routage 3D présenté précédemment détermine un itinéraire 3D composé de routes en 2D, en utilisant RILM. L'une des décisions les plus importantes dans chaque couche est de déterminer *le meilleur* nœud V_{To} pour le routage inter-couche. Le meilleur nœud vertical dépend de nombreux critères (le nombre des étapes ou la distance géométrique, la congestion le long de la route en 2D, les congestions dans le nœud vertical respectif etc.). Un algorithme de base pour la sélection de nœuds verticaux V_{To} (dénommés V_{ToUp} et V_{ToDown} dans l'algorithme Algorithm 8-1) est présenté dans cette sous-section, pour des couches 2D avec n'importe quelle topologie. La deuxième partie est dédiée à la fonction de l'adaptabilité de l'algorithme, en montrant comment l'attribution de nœud vertical peut être optimisée compte tenu des différents critères.

8.5.2.1 Construction de VNT dans les couches 2D

L'objectif principal de la phase de construction de VNT (arbre enraciné dans un nœud vertical) est d'affecter au départ à chaque nœud de la couche deux nœuds verticaux dans la même couche (V_{ToUp} et V_{ToDown}), pour le transfert inter-couche de messages. Le mécanisme de VNT peut être utilisé pour attribuer des nœuds verticaux dans n'importe quelle topologie 2D régulière ou irrégulière. La phase d'attribution des nœuds verticaux doit être exécutée durant la phase d'initialisation du système (mais elle peut aussi être ré-exécutée plus tard en cas de probabilité plus élevée de défaillance pendant la durée de vie). Le processus d'affectation des nœuds verticaux est lancé au niveau des nœuds verticaux et se propage vers les autres nœuds, en utilisant une technique de commérage (gossiping) [PGC06] contrôlée. Grâce à ce processus, les arbres enracinés dans les nœuds verticaux et appelés VNT sont construits. Les VNTs résultants sont maintenus afin d'assurer la possibilité de reconfigurations locales (voir les sous-sections 8.5.2.2 et 8.5.3).

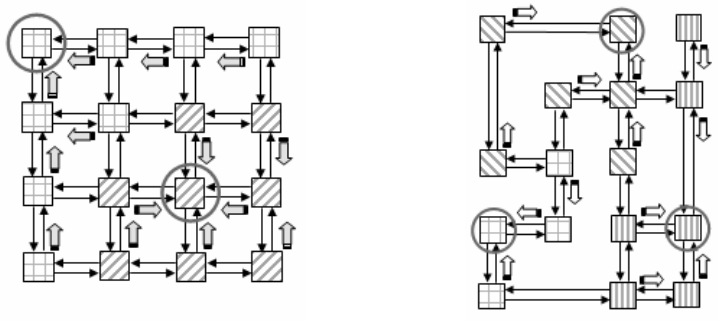
Deux strates de VNT doivent être créées dans chaque couche du NoC 3D, une pour chaque sens vertical. Par souci de simplicité, une seule direction est considérée dans les paragraphes suivants, c'est-à-dire une seule strate de VNTs.

Les liens verticaux d'une couche sont susceptibles d'être moins nombreux que les nœuds de cette couche. Par conséquent, plusieurs nœuds utiliseront le même lien vertical (on dit que les nœuds sont *attachés* au lien vertical respectif). Un VNT contient tous les nœuds attachés au nœud racine. La mise en œuvre la plus économique d'un arbre est de maintenir à chaque nœud un *pointeur vers le nœud parent*.

Les messages échangés pour la construction et de reconfiguration des VNTs sont simplement appelés *messages VNT*. Pour la phase de construction de VNTs, seul un type de messages VNT est utilisé, les *messages d'attachement*, désigné par *mA*. Ils sont aussi appelés messages de *connexion* ou de *joint*. Les messages mA sont envoyés uniquement aux voisins d'un nœud. La phase de construction est initiée par des nœuds verticaux, qui se sont fixés comme des racines VNT et envoient ensuite des messages *mA* à leurs voisins. Dès la réception d'un *mA*, un nœud qui n'est pas attaché sera joint à la racine indiquée dans le message. Il rejoint le même VNT que l'expéditeur du message, qui est l'un des nœuds voisins du nœud courant. L'expéditeur devient le parent du nœud courant et le nœud actuel devient le fils de l'expéditeur dans le même VNT. Le nouveau nœud joint, à son tour, envoie des *mA*s à ses voisins. La phase de construction de VNTs est un

processus fini, car une fois attaché à un VNT, les nœuds cessent d'envoyer des messages VNT.

Des exemples de strates VNT pour une seule direction verticale sont représentés dans la Fig. 8-9, pour différentes topologies non-orientées. Les nœuds verticaux sont entourés et les nœuds du même VNT sont remplis de la même texture. Des flèches pleines indiquent le parent VNT de chaque nœud 2D non-défectueux, ce qui représente les arcs du VNT.



a. VNTs dans une topologie en maille b VNTs dans une topologie irrégulière

Fig. 8-9 Exemples de VNTs

Initialement, tous les nœuds sont à l'état *nU*, c'est-à-dire *seuls* ou *non attachés* à un nœud vertical. Après la phase de construction de VNTs, tous les nœuds sont dans l'état *nA*, c'est-à-dire attaché ou connecté ou joint dans un VNT, grâce à l'hypothèse de non-partitionnement de la couche.

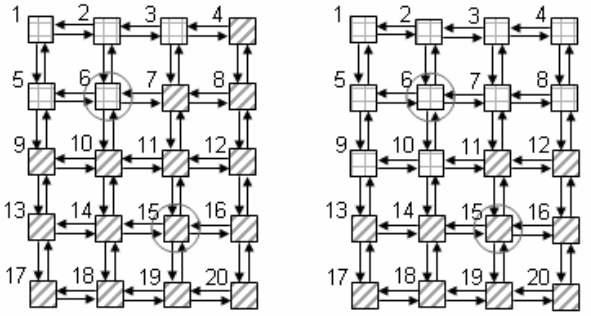
8.5.2.2 Reconfiguration pour l'adaptabilité

Dans les systèmes réels, plusieurs facteurs peuvent contribuer à la répartition inégale des nœuds dans les VNTs. Une fonction de poids supplémentaire dans l'algorithme de VNT peut équilibrer les VNTs selon les critères sélectionnés. La fonction de poids est également une solution à l'attachement non-optimal du point de vue de la longueur du chemin de routage. Un *poids* est associé à chaque attachement d'un nœud dans un VNT, indiquant la rentabilité de l'attachement respectif par le critère choisi. L'attachement optimal est celui de poids minimal ou maximal.

La phase d'attachement VNT peut être complétée par une telle fonction du poids. Un champ de poids supplémentaire doit être conservé à chaque noeud. La connexion elle-même n'est pas modifiée, mais elle est conditionnée par la relation entre les poids courant et potentiel de l'attachement du noeud dans le VNT courant et dans le nouveau VNT possible, respectivement.

Des reconnexions peuvent également être faites dans le même VNT, mais par un autre parent. En ce qui concerne le routage ou l'équilibrage de trafic sur les nœuds verticaux, une reconnexion dans le même VNT ne présente aucun intérêt. Toutefois, cela peut être utile à d'autres fins, telles que la propagation plus rapide des informations de déconnexion dans le VNT, dans le cas d'une fonction de poids égal au nombre d'arrêtes de la racine (puisque la hauteur du VNT est réduite au minimum).

Dans la Fig. 8-10, une configuration initiale (a) et une configuration équilibrée (b) de VNT sont représentées. La fonction de poids utilisée pour équilibrer le VNT est le nombre d'arcs du nœud vertical.



a. Configuration initiale de VNTs b. VNTs équilibrés

Fig. 8-10 Différents équilibrations de VNTs

Après chaque reconnexion en raison de la fonction de poids, le nœud envoie des messages *mA* à ses voisins pour les informer de la reconnexion. Les fils du nœud mettent à jour leur fonction de poids et peuvent éventuellement décider par la suite d'autres reconnexions.

D'autres types de fonction de poids peuvent être implémentés dans l'algorithme de construction de VNTs, selon les exigences du système. Un autre exemple de fonction de poids est le nombre de nœuds dans le VNT. Toutefois, pour son calcul, des informations centralisées sur l'ensemble du VNT sont nécessaires.

Dans certains cas, une répartition inégale des nœuds sur les liens verticaux peut être souhaitable. C'est le cas de nœuds qui engagent (ou simplement transfèrent d'autres niveaux) des charges de trafic inégal sur les liens verticaux. L'adaptabilité peut être atteinte en transférant des charges de trafic plus faibles sur les liens avec des plus faibles nombres de TSVs fonctionnels (sans faute), par exemple. En général, une fonction adaptée de poids VNT, obtenue par l'analyse de la charge de trafic vertical réel induit par les nœuds de la couche, peut «équilibrer» les VNTs.

8.5.3 Reconfiguration en présence des défaillances

En cas de pannes, les systèmes se reconfigurent afin de restaurer la connectivité du réseau de communication [AN05]. Les routes en 2D ne sont pas les seules touchées par les pannes de la couche. En même temps, en cas de défaillance, les VNTs ne sont plus cohérents. Recréer les VNTs dans la configuration de la nouvelle topologie avec l'algorithme Algorithm 8-1 est une solution possible pour obtenir des VNTs cohérentes. Toutefois, la reconfiguration VNT présentée dans cette sous-section peut être faite localement dans la région affectée par la panne (ou la défaillance). Du point de vue des nœuds affectés par la panne, la reconfiguration se fait en deux phases : la *déconnexion* (ou *détachement*) de VNT, puis la *reconnexion* (ou la *rejonction* ou le *rattachement*) dans un VNT.

8.5.3.1 Détachement de VNT

Le VNT est composé de nœuds et des chemins utilisés par les messages *mA* pendant la phase d'attachement des nœuds. En cas de défaillance le long de ces chemins, la structure du VNT est considérée comme altérée. Les nœuds de tous les (sous)VNTs du nœud ou lien (d'un nœud parent à un nœud fils) défaillant sont *affectés par la défaillance*, *AF* (*affected by failure*). Les racines de ces (sous)VNTs sont des voisins du nœud / lien défaillant. Les voisins d'un élément défaillant sont conscients de cette défaillance. Par conséquent, la phase de détachement de nœuds affectés est initiée par les voisins de

l'élément défaillant, appelés *initiateurs de la phase de détachement*, *DPI* (*detaching phase initiators*). Ils commencent la phase de détachement par leur propre déconnexion du VNT. Ensuite ils envoient des messages de *détachement* à leurs voisins de premier ordre. Les messages de *détachement* (ou de *déconnexion*) représentent le deuxième type de messages VNT et sont désignés par *mD*. De la même façon que les messages *mA*, les messages *mD* sont envoyés uniquement aux voisins de l'expéditeur.

Si le nœud qui reçoit le message *mD* est joint et l'expéditeur du *mD* est son parent pour le sens vertical respectif, le nœud se détache : il passe en état *nD* (*noeud déconnecté*) et envoie des messages *mD* à ses voisins.

Il est montré que le (sous)VNT précis qui est enraciné dans le noeud initiateur (DPI) se détache, c'est-à-dire, tous les nœuds affectés pas la défaillance, *AF*s, se détachent.

8.5.3.2 Rattachement dans VNT

La deuxième phase de la reconfiguration est celle de rattachement (reconnexion). Elle est initiée par des nœuds attachés qui reçoivent des messages *mD*. Ces nœuds sont appelés *initiateurs de la phase de rattachement*, *RPI* (*reattaching phase initiators*). Ils sont indirectement touchés par la défaillance, c'est-à-dire qu'ils n'ont pas besoin de se déconnecter et se reconnecter, mais participent au processus de reconfiguration. Les *RPI*s sont des voisins du (sous)VNT détaché et limitent l'expansion de la phase de détachement. Ils sont plutôt des initiateurs potentiels de la phase de rattachement de leurs nœuds voisins détachés.

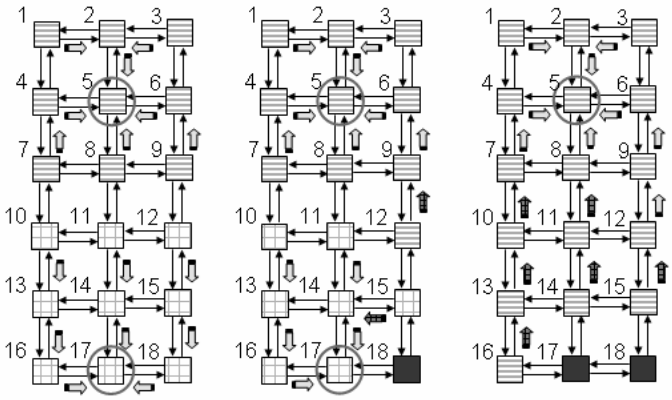
La phase de rattachement peut être faite différemment pour les topologies non-orientés (graphes avec seulement des arêtes bidirectionnelles) et celles mixtes (des arcs unidirectionnels y existent aussi). Un algorithme plus simple peut être mis en œuvre pour le cas non-orienté. Toutefois, celui proposé pour les graphes mixtes est général et fonctionne pour les deux types de topologies (graphes non-orientés et graphes mixtes).

8.5.3.2.1 Rattachement au VNT dans une topologie de graphes non-orientés des couches 2D

Dès la réception d'un message *mD*, un *RPI* envoie des messages *mA* à ses voisins. C'est le lancement de la phase de rattachement.

Dès la réception d'un message *mA*, un nœud dans l'état *nD* (nœud déconnecté) se comporte comme un nœud initialement non connecté (état *nU*) dans la phase de construction de VNTs. Ainsi, l'algorithme doit être complété afin d'inclure ce cas.

Dans la Fig. 8-11, un exemple de configuration de VNT initial (pour un seul sens) et les configurations obtenues après des pannes successives des nœuds est présenté pour un réseau de maillage régulier.



a. VNTs initiaux b. Après la panne du nœud 18 c. Après la panne du nœud 17

Fig. 8-11 Construction et reconfiguration des VNTs en maillage régulier

Les nœuds de 1 à 9 appartiennent au VNT_5, enraciné dans le nœud 5 et les nœuds de 10 à 18 appartiennent au VNT_{17}, enraciné dans le nœud 17 (Fig. 8-11.a). Les nœuds défectueux sont en noir. Il n'y a pas des nœuds défectueux dans la configuration initiale. Si le nœud 18 tombe en panne, les nœuds de son sous-arbre (15 et 12) se détachent. Ils rejoignent les VNTs par certains de leurs voisins non-fautifs : le nœud 15 rejoint le VNT_{17} par son voisin 14 et le nœud 12 rejoint le VNT_5 par son voisin 9. Après une défaillance supplémentaire du nœud 17, tous les nœuds de l'ancien VNT_{17} se reconnectent dans le VNT_5. La phase de reconnexion est initiée par les nœuds de la frontière du VNT_5 qui sont atteint par des messages *mD* venant du VNT_{17}. Ces nœuds sont : 7, 8 et 12. Ils répandent des messages *mA* à leurs voisins et, ainsi, les nœuds voisins détachés rejoignent le VNT_5. Ces nœuds sont les suivants : 10, 11 et 15. Après le rattachement, ils propagent des *mA*s, de sorte que la reconnexion se poursuit jusqu'à ce que tout le VNT détaché soit reconnecté.

8.5.3.2.2 Rattachement au VNT dans une topologie de graphes mixtes des couches 2D

Les topologies de graphes mixtes présentent des cas de reconfiguration plus générale et plus complexe. Dans ces graphes, il est possible que des liens des initiateurs de la phase de rattachement (*RPI*s) vers les nœuds déconnectés n'existent pas toujours. Par conséquent, les messages *mA* peuvent ne pas atteindre les nœuds détachés. Dans de tels cas, l'initiation de la phase de reconnexion se fait par un troisième type de messages VNT, notée par *mR* (message de *reconnexion*, *rattachement* ou *rejointe*). Ces messages ne sont pas envoyés aux voisins du nœud, mais sont dirigés vers le nœud déconnecté ; ils sont envoyés sous forme de messages ordinaires. Dans les topologies de graphes non-orienté, les messages *mA* suffisent pour atteindre tous les nœuds d'un VNT à partir de n'importe quel nœud qui se reconnecte. Pour les topologies de graphes mixtes, les messages *mR* sont utilisés pour diriger la reconnexion vers la racine de l'arbre, puisque les liens vers des nœuds fils peuvent être unidirectionnels. L'envoi de messages *mA* est toujours utilisé dans l'algorithme, et ils complètent les messages *mR*. Etant donné que les graphes non-orientés sont un cas particulier de graphes mixtes, cet algorithme de rattachement fonctionne aussi pour les graphes non-orientés.

Dans les deux cas (graphes non-orientés et mixtes), tous les nœuds détachés du VNT auquel ils appartenaient sont reconnectés, en se joignant à un autre (ou au même) VNT, par l'exécution des algorithmes de rattachement proposés.

Plusieurs simulations montrent la localisation des reconfigurations après défaillances. Cependant, la propagation de la reconfiguration dépend de plusieurs facteurs, tels que la multiplicité et le motif des défauts, le taux et le motif des liens verticaux existants dans la topologie, les conditions d'exécution dans le NoC, l'équilibrage des VNTs etc. En moyenne, la reconfiguration dure moins de 50% de la durée de reconstruction des VNTs. En ce qui concerne le routage 3D à l'aide de RILM, une latence acceptable est requise, même pour un petit nombre de liens verticaux disponibles. Les facteurs qui influencent le temps de latence du routage sont la géométrie de la topologie, le nombre de liens verticaux disponibles et les configurations des VNTs.

8.6 Récupération des erreurs temporaires aux niveaux des couches de lien et de transport

Cette partie présente des solutions tolérantes aux fautes pour faire face à des fautes temporaires dans les NoCs, en agissant à la fois au niveau des couches de liaison de données et celui du transport. La couche de l'application est également impliquée, car les techniques de tolérance aux fautes peuvent être configurées pour respecter le niveau de la QoS des données transmises. Une bibliothèque générique de modules de tolérance aux fautes implantant plusieurs types de codes détecteurs / correcteurs d'erreurs (*error detection / correction codes, EDC/ECC*) a été développée. Le but de ce travail est de faciliter le choix du compromis entre l'approche tolérante aux fautes la plus adaptée pour le niveau de tolérance ciblé, en fonction de la QoS, et les surcoûts encourus en termes de surface, consommation d'énergie et latence.

8.6.1 Mécanismes de récupération d'erreurs

Sur la base des codes de détection et correction d'erreurs, deux principaux mécanismes de *récupération d'erreurs* peuvent être utilisés :

- La *correction d'erreurs*, en utilisant des codes correcteurs d'erreurs ;
- La *retransmission de la donnée*, en utilisant des codes détecteurs d'erreurs.

Dans les NoCs, ces mécanismes peuvent être appliqués :

- *Commutateur-à-commutateur, S2S (switch-to-switch)* ou au niveau du *lien*, c'est-à-dire à chaque nœud le long du chemin des données ;
- *Bout-à-bout, E2E* (*end-to-end*) ou au niveau du *transport*, c'est-à-dire entre les *interfaces réseau* (*NI*s, *network interfaces*) des nœuds source et destination.

Les tableaux suivants reprennent les caractéristiques des mécanismes de base de récupération d'erreurs (la correction et la retransmission), appliquées au S2S ou E2E.

Les schémas S2S traitent les erreurs cumulées le long du chemin. Dans les régimes E2E, la multiplicité d'erreur peut dépasser la capacité des EDC / ECC (Table 8-1). Toutefois, un prix plus élevé est payé pour la mise en œuvre des S2S, en termes de surcoût en surface et puissance, car les blocs de contrôle d'erreurs doivent être ajoutés pour chaque paire de nœuds voisins. Dans les schémas E2E, ceux-ci sont présents uniquement aux NIs. La largeur du tampon de retransmission peut être plus étroite pour les schémas S2S, vu que les tampons stockent des données originales (vs. données codées dans les schémas E2E).

Table 8-1 Efficacité du contrôle d'erreurs dans les schémas commutateur-à-commutateur par rapport à bout-à-bout

Critère/schéma	**S2S**	**E2E**
Traite les erreurs cumulées le long du chemin	*oui*	*non*
Surcoût en surface et consommation	*grand*	*petit*
Largeur du tampon intermédiaire	*large*	*étroite*

La latence des mécanismes de recouvrement d'erreur dans les quatre cas de base de récupération d'erreurs est présentée dans la Table 8-2. Une retransmission S2S prend 3 cycles d'horloge, alors que la correction d'erreurs S2S ne prend que 1 cycle. La correction a la même latence, y compris pour le cas E2E avec n'importe quel nombre de hops, mais il ne traite pas les erreurs cumulées le long du chemin. La latence de retransmission E2E ne peut pas être estimée simplement, car elle dépend de plusieurs facteurs, comme la longueur du trajet entre les deux nœuds d'extrémité et de la charge de trafic sur cette voie, tant pour la demande de retransmission et la retransmission des données elle-mêmes.

Table 8-2 La latence des mécanismes de récupération d'erreurs

Latence	**S2S**	**E2E**
Retransmission	*3 cycles d'horloge / lien*	*dépend du chemin, charge de trafic etc.*
Correction	*1 cycle d'horloge / lien*	*1 cycle d'horloge*

Les codes de correction d'erreurs sont plus complexes que les codes de détection d'erreurs, et donc, les coûts correspondants sont plus élevés. D'autre part, la retransmission implique l'existence de tampons pour stocker des données pour d'éventuelles retransmissions futures. Ainsi, la surface globale pour la retransmission est plus élevée que dans le cas de la correction (Table 8-3). Toutefois, en payant le prix du tampon de retransmission, une plus grande efficacité du système de tolérance aux fautes est atteinte, puisque la récupération est possible aussi pour des erreurs qui ne peuvent pas être corrigées par le code correcteur d'erreurs.

Table 8-3 Surcoût en surface de la correction et de la retransmission

Schéma	Surcoût en surface
Retransmission	*grand*
Correction	*petit*

L'efficacité des quatre schémas de base de la récupération d'erreurs est synthétisée dans le tableau Table 8-4. Lorsque des erreurs sont susceptibles de se produire uniformément réparties le long du chemin, les schémas S2S sont plus efficaces, car ils traitent chaque erreur quand cela se produit. La correction d'erreurs S2S traite des erreurs d'une multiplicité plus basse (limitée par la capacité du code de correction d'erreurs), uniformément réparties le long du chemin. D'autre part, la retransmission S2S est plus efficace, surtout quand il y a beaucoup d'erreurs uniformément répartis le long du chemin, qui ne peuvent pas être traitées par le code de correction d'erreurs.

Table 8-4 Efficacité de la correction et de la retransmission

Efficacité	**S2S**	**E2E**
Retrans-mission	- *Erreurs de haute multiplicité* - *Erreurs uniformément réparties le long du chemin*	- *Erreurs de haute multiplicité* - *Erreurs non uniformément réparties le long du chemin*
Correction	- *Erreurs de faible multiplicité* - *Erreurs uniformément réparties le long du chemin*	- *Erreurs de faible multiplicité* - *Erreurs non uniformément réparties le long du chemin*

Si les erreurs ne sont pas uniformément réparties le long du chemin, le contrôle d'erreurs S2S est inutile. Pour gagner en temps de latence, les schémas E2E sont préférés. Comme dans le cas des schémas S2S, la retransmission E2E peut traiter plusieurs types d'erreurs que l'approche de correction. Cette limitation est donnée par la capacité de détection / correction d'erreurs du code sélectionné.

8.6.2 Schémas de contrôle d'erreurs

Deux systèmes principaux de contrôle d'erreurs correspondent aux mécanismes de reprise d'erreurs de base : la correction d'erreurs et de retransmission des données. Dans de nombreux cas, certains types d'erreurs sont plus susceptibles de se produire, tandis que d'autres types (le plus souvent celles d'une multiplicité élevée et un motif irrégulier) sont plus rares. Par conséquent, un compromis entre l'efficacité du schéma de tolérance aux fautes et son coût doit être pris en considération. La solution à cette question est un schéma *hybride*, qui combine la correction d'erreurs avec la retransmission.

En considérant des applications avec des QoS différentes, nous proposons le schéma de contrôle d'erreurs *sélectif*. Le schéma *sélectif* détecte les erreurs et peut éventuellement les corriger. Dans ce schéma, la correction est faite sur la base des informations latérales envoyées sur le lien à coté des éléments de données (les *flits*). Ces informations sont ajoutées par l'interface réseau, sous le contrôle des applications ou des pilotes des IP du SoC.

Les caractéristiques des schémas de contrôle d'erreurs mentionnés ci-dessus sont synthétisées dans les lignes suivantes :

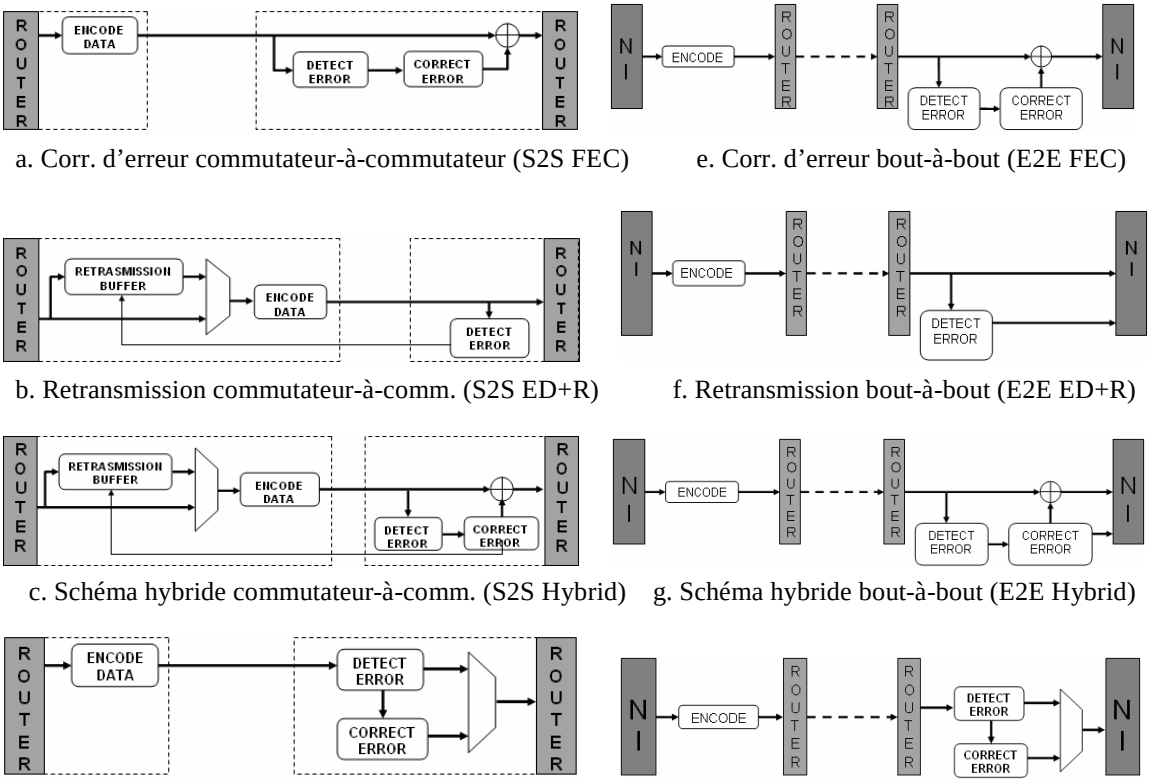
◊ *Correction, FEC (Forward Error Correction*) : les données reçues sont toujours corrigées (uniquement les codes de correction d'erreurs sont utilisés : Hamming SEC/DED, Hsiao).

◊ *Détection d'erreurs et retransmission, ED+R* (*Error Detection and Retransmission*) : les données sont renvoyées lorsque des erreurs sont détectées (des codes de détection d'erreurs sont utilisés : la parité, le code Hamming).

◊ *Hybride* : corrige autant d'erreurs que possible, sinon demande la retransmission, pour améliorer la capacité de correction d'erreurs (des codes correcteurs d'erreurs sont utilisés).

◊ *Sélectif* : corriger les erreurs uniquement lorsque c'est explicitement demandé par les niveaux supérieurs (des codes correcteurs d'erreurs sont utilisés).

Tous ces schémas peuvent être appliqués soit entre chaque paire de commutateurs adjacents (ou routeurs) ou seulement entre la source et la destination de chaque message, comme représenté dans la Fig. 8-12. Les schémas commutateur-à-commutateur (S2S) sont représentés dans la Fig. 8-12.a-d et ceux bout-à-bout (E2E), dans la Fig. 8-12.e-h.



a. Corr. d'erreur commutateur-à-commutateur (S2S FEC) e. Corr. d'erreur bout-à-bout (E2E FEC)

b. Retransmission commutateur-à-comm. (S2S ED+R) f. Retransmission bout-à-bout (E2E ED+R)

c. Schéma hybride commutateur-à-comm. (S2S Hybrid) g. Schéma hybride bout-à-bout (E2E Hybrid)

d. Schéma sélectif commutateur-à-comm. (S2S Selective) h. Schéma sélectif bout-à-bout (E2E Selective)

Fig. 8-12 Schémas de contrôle d'erreurs

- Dans le schéma *S2S FEC* (Fig. 8-12.a), les modules ajoutés au lien simple entre deux routeurs/commutateurs sont : l'encodeur (du côté de l'expéditeur), le détecteur et le correcteur (du côté du récepteur).
- Dans la Fig. 8-12.b, le schéma *S2S ED+R* est représenté. Le module d'encodage est également présent dans la partie émission, et il est complété par un tampon de retransmission. Lorsque de nouvelles données sont envoyées, elles sont également stockées dans le tampon de retransmission. En cas de demande de retransmission, les données à envoyer sont récupérées dans le tampon de retransmission. Au récepteur, un module de détection a pour rôle de détecter l'erreur et de demander la retransmission des données en cas d'erreurs. Ainsi, le canal de communication est complété par la demande de retransmission, dans le sens inverse.
- Le schéma *S2S hybride* combinant la correction d'erreurs et la retransmission est présenté dans la Fig. 8-12.c. Outre le module de détection d'erreurs, le schéma *hybride* contient des modules pour la retransmission et aussi pour la correction d'erreurs. En cas d'erreurs, le schéma *hybride* effectue la correction lorsque cela est possible et la retransmission autrement. Puisque la retransmission est plus lente que la correction, le schéma *hybride* est plus rapide que la retransmission pure. La capacité d'adaptation et la réduction de la latence du schéma *hybride* sont toutefois payées en surface.
- Dans la Fig. 8-12.d, le schéma *S2S sélectif* de correction d'erreurs est affiché. Les données (*flits*) envoyées sur les fils sont codées avant de quitter l'interface en amont.

La détection et la correction éventuelle d'erreur sont mises en œuvre avant que les données arrivent dans l'interface en aval (Fig. 8-12.d). Lorsque des erreurs sont détectées, les informations latérales envoyées sur les lignes de commande indiquent si une correction doit être apportée ou non. Si la correction n'est pas nécessaire, parce que soit il n'y a pas d'erreurs détectées soit les erreurs seront corrigées par les couches

supérieures, les flits arrivent à l'interface aval avec le délai induit par la détection. Lorsque les erreurs détectées doivent être corrigées, un délai supplémentaire est ajouté par l'opération de correction.

Sur les sorties des modules de détection et de correction du schéma *sélectif* (Fig. 8-12.d) il y a respectivement des données corrigées et non corrigées. Ces sorties sont multiplexées avant que les données arrivent à l'interface en aval. Le multiplexeur est commandé par une bascule qui est mise à 1 par une décision de correction et remise à 0 lorsque la transmission se termine. Lorsque la bascule est activée, la sortie du module de correction est transmise à l'interface aval. Dans l'autre cas, les données non corrigées à la sortie du module de détection sont transmises. Par conséquent, une fois qu'un flit est corrigé, tous les flits suivants sont retardés d'un cycle d'horloge supplémentaire et l'omission de l'étape de correction devient possible après la fin de la transmission. Ainsi, le schéma *sélectif* a une latence similaire avec celle de FEC, lorsque la correction est nécessaire, et plus faible autrement.

Les schémas *E2E* (Fig. 8-12.e-h) fonctionnent selon le même principe que les schémas *S2S*, mais seulement aux nœuds d'extrémité (c'est-à-dire entre la source et la destination des messages). Dans les schémas de bout-à-bout, les mêmes modules sont toujours présents à chaque routeur connecté à une interface réseau (*NI*), mais ils ne fonctionnent pas à moins qu'ils appartiennent au nœud source ou au nœud destination du message courant, en réduisant ainsi la latence du message. Pour les demandes de retransmission, à la différence du cas S2S, des fils dédiés ne doivent pas être ajoutés dans le NoC. Au lieu d'avoir du matériel supplémentaire, la demande de retransmission est réalisée par des messages qui transitent par le NoC.

Dans tous les schémas de contrôle d'erreurs, les **fils de contrôle** contenant des informations critiques sont protégés grâce à la redondance modulaire triple (**TMR**) et le vote majoritaire.

Pour la **détection** d'erreurs, des codes de détection d'erreurs simples et doubles sont utilisés. Pour la **correction**, des codes de correction d'erreurs simples sont utilisés.

Pour faire face à des **erreurs multiples**, des techniques qui appliquent des codes simples détecteurs / correcteurs sur des blocs ou des groupes entrelacés de bits sont utilisées pour obtenir des codes avec des distances de Hamming plus grandes (comme indiqué dans la sous-section 6.2.3.1, pour les codes de parité). La façon dont les groupes sont formés influe sur la distribution des erreurs dans les motifs. Par exemple, la diaphonie induite par des erreurs multiples affecte habituellement les fils adjacents et l'entrelacement des codes SEC (c.-à-d. des fils adjacents sont dans des groupes différents) corrige toutes ces erreurs.

Néanmoins, tout autre type de codes à distance plus élevée peut également être utilisé à la place du bloc/entrelacement des codes simples.

8.6.3 Bibliothèque de codes linéaires et environnement de simulation

Sur la base des propriétés communes des codes linéaires, une bibliothèque de codes a été construite. Elle se sert d'une base de données des matrices *G* et *H* pour plusieurs types de codes et tailles. Toutefois, d'autres codes peuvent être ajoutés à la bibliothèque en ajoutant simplement leurs matrices *G* et *H* dans la base de données.

Pour tout code linéaire, les modules de codage, de détection et de correction d'erreurs sont générés automatiquement, en utilisant les matrices *G* et *H*, et les formules des codes linéaires (voir la section 6.2 pour plus de détails).

La génération des liens tolérants aux fautes est représentée dans la Fig. 8-13. Pour chaque taille des données, le type de code doit être précisé (code d'identification, *n*, *k*), ainsi que le nombre de groupes (pour le codage par bloc ou entrelacé). Ces informations sont utilisées pour sélectionner les matrices appropriées dans la base de données. Le triplage et le vote sont utilisés pour générer des signaux de commande tolérants aux fautes.

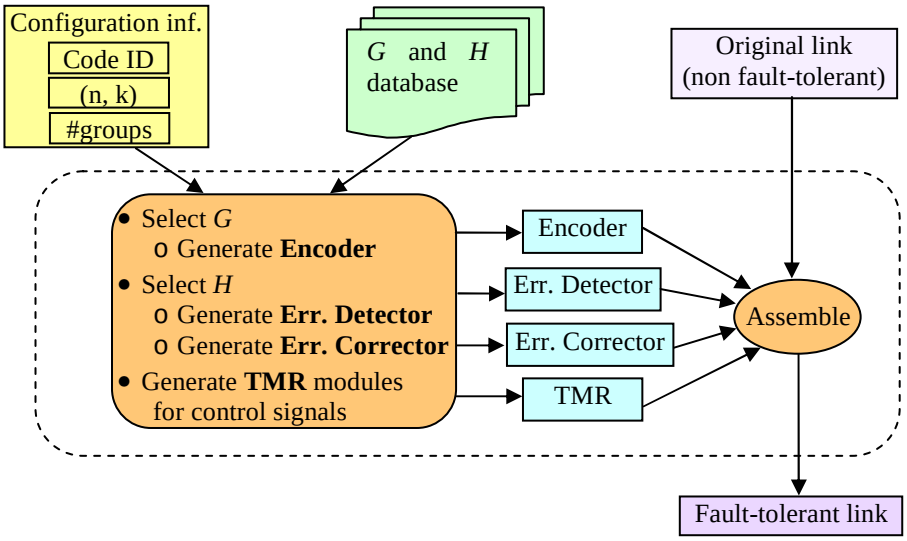


Fig. 8-13 Génération des liens tolérants aux fautes

Le lien tolérant aux fautes est obtenu à partir du lien original, en ajoutant dans les interfaces en amont et en aval des modules de tolérance aux fautes et en remplaçant les fils de commande avec leurs homologues tolérants aux fautes.

Pour des simulations dynamiques des liens, un **générateur de trafic** et un **absorbeur de trafic** sont raccordés sur le lien en cours d'évaluation. Le générateur de trafic génère un trafic en rafales, c'est-à-dire un flit est dans injecté dans le lien à chaque cycle d'horloge. Le consommateur de trafic récupère les flits du lien dès qu'ils sont disponibles.

8.6.4 Latence et surcoût en surface

L'amélioration de la fiabilité de la communication obtenue par des méthodes de tolérance aux fautes induit des coûts supplémentaires dans le silicium : la surface et la puissance dissipée par les modules additionnels et le nombre de fils. Le nombre de fils est également un paramètre important, surtout lorsque le lien est un lien vertical dans un NoC 3D, parce que les faisceaux verticaux ont des empreintes larges sur les couches. Dans la technologie actuelle, les TSVs ont un pas de dizaines de µm et 30 liens de 64 bits chacun ont une superficie totale de 0,7 μm^2, soit l'équivalent de la surface d'un bloc IP classique. Le nombre des fils supplémentaires donne une approximation de la pénalité en surface induite par l'augmentation des capacités de correction d'erreurs (c.-à-d. les bits de contrôle supplémentaires).

Pour évaluer le coût de la tolérance aux fautes, le modèle RTL du lien ayant une meilleure fiabilité est synthétisé dans le processus de 65 nm de ST Microelectronics.

Les surcoûts en surface et puissance pour les quatre schémas de contrôle d'erreurs (*FEC*, *ED+R*, *hybride* et *sélectif*) sont évalués. Plusieurs tailles de données et des

configurations de codage par bloc/entrelacé sont prises en considération. Le schéma *hybride* a les surcoûts les plus élevés, car il contient à la fois des modules de correction d'erreurs et de retransmission. Il est suivi par le schéma *ED+R*, en raison des tampons de retransmission qu'il contient aussi. Le *FEC* a le plus faible surcoût. Le schéma *sélectif* a un surcoût faible, comparable à celui du schéma *FEC*. En ce qui concerne la puissance, elle est supérieure, plus proche de celle du schéma *ED+R*. Ceci est expliqué par la présence de la logique de sélection.

L'analyse matérielle du schéma *sélectif* proposé montre que les surcoûts en surface et en puissance dissipée pour un lien ont de faibles variations pour des flits de tailles petites et moyennes. Le nombre de fils dépend du nombre de groupes dans le schéma de codage et il a un impact sur l'empreinte de l'interconnexion verticale, puisque plusieurs groupes signifient plusieurs fils verticaux. En même temps, plusieurs groupes augmentent les capacités de correction d'erreurs et, comme l'analyse la montre, avec des pénalités faibles en surface et en puissance.

Le surcoût en surface du schéma *sélectif* de correction d'erreurs est également évalué par rapport au NoC. Un NoC 3D est utilisé pour cette évaluation. Trois types de protection sont envisagées : commutateur-à-commutateur, bout-à-bout et sur les liens verticaux seulement. Des surcoûts plus faibles sont obtenus au niveau du NoC. Parmi les trois configurations, l'implantation du schéma S2S est la plus consommatrice en surface et il est recommandé pour des taux d'erreur élevés. Ensuite, la protection E2E n'a besoin que de moins de 20% de surcoût en surface. La troisième configuration est particulièrement intéressante pour protéger les liens verticaux des erreurs qui se produisent dans les TSVs. Une connectivité verticale complète est considérée dans cette estimation et tous les liens verticaux du NoC sont protégés. Le surcoût en surface représente moins de 10%. En outre, les surcoûts au niveau du système à base de NoC (y compris aussi les unités de calcul et les interfaces réseau) sont encore plus petits. Par conséquent, même si au niveau du lien les surcoûts sont assez importants, du point de vue du système, elles sont faibles, voire négligeables dans certains cas.

8.7 Conclusions et travaux futurs

Les puces modernes sont de plus en plus complexes et hétérogènes ; ils ont différentes exigences en matière de qualité de service de l'application et peuvent être réalisés dans la technologie d'intégration 3D. Dans le même temps, les SoC sont sujets à des défaillances causées par des fautes transitoires, intermittentes et permanentes. Ces fautes peuvent provenir des dimensions diminué des composants électroniques, peuvent être produites en cours de fabrication ou, dans la durée de vie, peuvent être combinées avec des facteurs environnementaux (radiations, interférences, etc.).

L'objectif général de cette thèse est de traiter des méthodes de tolérance aux fautes qui prennent en compte la complexité des puces modernes et leur hétérogénéité, afin de minimiser les surcoûts de la tolérance aux fautes et de réaliser des solutions extensibles. De nombreuses approches de tolérance aux fautes ont été proposées dans le passé, à tous les niveaux du protocole de communication et à tous les composants et sous-composantes du NoC. Cependant, cette thèse n'est pas une vue d'ensemble de ces méthodes de tolérance aux fautes, mais se concentre sur trois aspects : restauration de l'application par points de contrôle, routage tolérant aux fautes et transmissions tolérantes aux fautes. Les solutions proposées dans la thèse répondent à la question générale formulée dans la deuxième section :

- **Question :** Comment faire pour atteindre des niveaux de fiabilité acceptable avec un impact réduit sur les exigences de qualité de service tout en offrant de l'adaptabilité aux spécificités des applications (intégrité des données, garantie de débit, meilleur effort) ?
  Réponse : La réponse à cette question est aussi générale que la question. La réponse peut être constituée en rassemblant les réponses aux questions portant sur chacun de ces mécanismes à tolérance aux fautes. Toutefois, la collaboration entre les différents mécanismes de tolérance aux fautes, au même niveau ou à différents niveaux, peut contribuer de manière significative à l'adaptabilité de la fiabilité du système en fonction des exigences de qualité de service, tout en minimisant les coûts de la latence, de la consommation d'énergie et de la surface.

La technique de restauration par points de contrôle (checkpoint and rollback recovery) a été initialement proposée pour récupérer rapidement des erreurs ou des défaillances / pannes de l'application qui sont causés par des fautes transitoires ou intermittentes. La technique peut également être utile en cas de défaillances ou pannes de nœuds du NoC, pour migrer et reprendre les tâches correspondantes sur d'autres nœuds non-défectueux. Une restauration rapide de l'application peut ainsi être atteinte. La technique de restauration de points de contrôle a été largement utilisée dans les macro-réseaux (par exemple les systèmes distribués), mais pas encore dans les NoCs, au meilleur de nos connaissances.

- **Question :** Comment l'incidence du mécanisme de récupération par points de contrôle sur la performance de NoC peut être évaluée et comment peut être améliorée la scalabilité de ce mécanisme ?
  Réponse : Afin d'évaluer l'impact des performances de cette technique dans les NoCs, nous avons développé des approches différentes de récupération par points de contrôle basées sur un modèle abstrait de l'application. Des améliorations des protocoles de base ont été également proposées pour augmenter leur scalabilité et minimiser leur impact sur les performances des applications, en tenant compte des spécificités des applications (tels que les tâches critiques ou temps réel et le partitionnement de la communication). Tous les protocoles sont indépendants de la topologie. Toutefois, un mécanisme de routage adapté aux spécificités de la topologie peut contribuer de manière significative à l'amélioration de la scalabilité des protocoles. En appliquant les améliorations de scalabilité aux niveaux de l'application et à celui du routage pour le protocole de prise de points de contrôle coordonnée, jusqu'à deux ordres de grandeur de réduction du temps de latence et des surcoûts en mémoire sont atteints pour les systèmes à 256 nœuds. Des réductions significatives de la durée de prise de point de contrôle (jusqu'à 5%) sont réalisées par l'exploitation de la partition de la communication. Le taux d'erreurs qui peut être toléré par l'approche bloquante de prise coordonnée de points de contrôle est plusieurs fois supérieure (2 à 6 dans nos simulations, en fonction de la charge de trafic) que dans l'approche non bloquante. Une approche unifiée bloquante – non bloquante de prise de point de contrôle est proposée, qui exploite les différentes exigences de qualité de service des applications.

Les architectures de NoC 3D exploitent à la fois les avantages des NoCs et de l'intégration 3D avec des couches empilées. Une topologie 3D avec une connectivité verticale partielle est une solution efficace pour l'utilisation d'un nombre réduit de TSV et qui s'adapte à l'hétérogénéité et les irrégularités des systèmes 3D actuels et futurs.

- **Question :** Comment faire pour router les messages dans un NoC 3D ayant une connectivité verticale partielle et des topologies de couches potentiellement différentes, avec un effort de conception, des surcoûts et des délais de mise sur le

marché minimaux, tout en assurant un routage optimal à l'intérieur de chaque couche ?

Réponse : RILM, un algorithme de routage inter-couches pour les NoCs 3D avec une connectivité verticale incomplète et peut-être différentes topologies régulières ou irrégulières dans les couches horizontales 2D a été proposé dans cette thèse. Des algorithmes déjà validés et adaptés aux topologies 2D peuvent être utilisés dans les couches 2D avec un surcoût minimal (seulement quelques champs supplémentaires doivent être maintenus à chaque nœud pour permettre le routage 3D). Outre de faire face aux irrégularités verticales de topologies de NoC 3D, l'algorithme de routage prend en compte les défaillances de routeurs et de liens du NoC. En fait, l'absence et la panne des liens verticaux sont traitées de la même manière. RILM permet la reconfiguration locale en cas de défaillance. La (re)intégration des liens verticaux et des routeurs réparés ou de rechange est également possible en utilisant le même mécanisme de reconfiguration. L'adaptabilité aux différentes exigences de qualité de service (en considérant par exemple les charges courantes de trafic dans le NoC ou la bande passante des liens verticaux ou leur débit) est possible grâce à une fonction personnalisable attachée à l'algorithme de routage. Le mécanisme de routage peut s'adapter à n'importe quel nombre de couches dans la pile 3D et peut tolérer un nombre élevé de défaillances : seulement deux liens verticaux (un vers le haut et un vers le bas) entre les couches adjacentes sont suffisantes pour assurer l'acheminement, tant que les couches ne sont pas cloisonnées.

Outre les défauts permanents, les transmissions dans VDSM sont affectées par des fautes transitoires et intermittentes.

- **Question :** Comment faire face aux fautes transitoires et intermittentes dans les liens de NoC ?

  Réponse : Le schéma de tolérance aux fautes proposé dans cette thèse est basé sur des codes de détection et correction d'erreurs. Afin de répondre à l'hétérogénéité de QoS des applications, des informations du niveau de l'application ou de transport sont exploitées pour le compromis fiabilité – latence. Plus précisément, pour les applications à débit garanti, la phase de correction est omise d'où un gain de temps de latence. Les erreurs éventuelles produites le long du chemin sont traitées au niveau du transport et de l'application. D'autre part, pour les applications de haute fiabilité, la correction peut être faite à chaque étape dans le chemin. Ainsi, les erreurs ne peuvent pas s'accumuler le long du chemin et compromettre ainsi l'intégrité des données. Pour les applications meilleur-effort, la correction peut être soit évitée ou effectuée, soit à chaque étape dans le chemin ou à la fin, en fonction de leur niveau de fiabilité requis ainsi que des autres applications partageant le NoC. Un léger surcoût est encouru par le schéma de tolérance aux fautes, car des codes efficaces sont mis en œuvre (comme Hamming et Hsiao). D'autres codes peuvent également être considérés et ajoutés à la bibliothèque que nous avons développée. Pour faire face à des erreurs multiples, le codage par blocs et groupes entrelacées peut être utilisé.

Travaux futurs

Pour les protocoles de recouvrement de l'application par points de contrôle, un algorithme de décision qui bascule dynamiquement entre les mécanismes avec blocage et sans blocage pour chaque phase de prise de points de contrôle peut être conçu et mis en œuvre pour les tâches qui acceptent les deux approches. L'instanciation et l'interchangeabilité dynamiques des configurations RMU, selon les schémas de communication dans les différentes phases de l'opération des NoCs, representent une autre direction future. L'évaluation de nos protocoles (dynamiquement configurés) sur différents

modèles de trafic d'application (pas uniquement le trafic uniforme) avec des caractéristiques non-statiques, en rafales, et des dépendances à long terme est une autre direction de travail à explorer. L'estimation des performances de tous ces protocoles en utilisant d'autres modèles d'application, des modèles repères (benchmarks) ou des applications réelles est une autre direction dans les travaux futurs.

Comme les travaux futurs dans le routage en NoC 3D, nous avons l'intention d'appliquer le mécanisme de routage pour d'autres topologies de couches 2D et évaluer son efficacité pour d'autres types de trafic. Différentes instanciations de la fonction de poids devront être évaluées. Pour obtenir une vue globale sur la charge de trafic sur un lien vertical, les VNTs dans les différentes couches doivent être considérés.

En ce qui concerne les systèmes de transmission tolérant aux fautes, des futures orientations de travail sont : l'achèvement de la bibliothèque avec d'autres codes, la comparaison de l'impact de ces schémas avec d'autres approches et l'évaluation de ces schémas pour des applications concrètes.

Comme travail à plus long terme, nous prévoyons de mettre en œuvre un NoC avec de la tolérance aux fautes multi-niveau, qui intègre toutes les solutions tolérantes aux fautes proposées.

Glossary

2D	Two Dimensional
3D	Three Dimensional
AF	(Node) Affected by Failure
AND	AND logic operation
BE	Best Effort
bps	Bits Per Second
C&R	Checkpoint and Rollback
CAC	Crosstalk Avoidance Code
CC	Coordinated Checkpointing
CIC	Communication-induced Checkpointing
CRC	Cyclic Redundancy Check
DED	Double Error Detection
DEP	Dependency
DFM	Design For Manufacturability
DPI	Detaching Phase Initiator
DRAM	Dynamic Random Access Memory
E2E	End-to-End
ECC	Error Correction Code
ED+R	Error Detection + Retransmission
EDC	Error Detection Code
FCR	Fault Containment Region
FEC	Forward Error Correction
FIFO	First In, First Out
FIT	Failures In Time
Flit	Flow-control unit
GALS	Globally Asynchronous, Locally Synchronous
GT	Guaranteed Throughput
IC	Integrated Circuit
LDPC	Low-Density Parity-Check
LFSR	Linear Feedback Shift Register
LLC	Logical Link Control
mA	message of Attaching
MAC	Media Access Control
mD	message of Detaching
MP-SoC	Multi-Processor System-on-Chip
mR	message of Reattaching
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
nA	node in the Attached state
nD	node in the Detached state
NI	Network Interface
NIR	Nodes Involved in Reconfiguration
NMR	N-Modular Redundancy
NoC	Network on Chip
nU	node in the Unattached state
OS	Operating System
OR	OR logic operation
P2P	Point to Point
PDA	Personal Digital Assistant
PE	Processing Engine/Element
QoS	Quality of Service
RAID	Redundant Array of Inexpensive Disks
RMU	Recovery Management Unit
RPI	Reattaching Phase Initiator
RTL	Register Transfer Level
S2S	Switch-to-Switch
SCI	State Consistency Information
SEC	Single Error Correction
SER	Soft-Error Rate
SiP	System-in-Package
SoC	System-on-Chip
SRAM	Static Random Access Memory
TLM	Transaction Level Model
TMR	Triple Modular Redundancy
TSV	Thru Silicon Via
UC	Uncoordinated Checkpointing
VDSM	Very Deep Sub Micron
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VNT	Vertical Node Tree
XOR	eXclusive OR logic operation

Bibliography

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, 2000.

[AER+99] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram S. Rao, Syed A. Husain, and Asanka Del Mel. An analysis of communication-induced checkpointing. Technical report, Austin, TX, USA, 1999.

[AG05] Anurag Agarwal and Vijay K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 19–28, New York, NY, USA, 2005. ACM.

[AHM93] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *In Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, 1993.

[AHW05] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Dealing with failures during failure recovery of distributed systems. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–6, New York, NY, USA, 2005. ACM.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[ALRV00] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Vytautas. Fundamental concepts of dependability, 2000.

[AM95] L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, and causal. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, page 229, Washington, DC, USA, 1995. IEEE Computer Society.

[AN05] Dimiter Avresky and Natcho Natchev. Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. *IEEE Trans. Comput.*, 54(5):603–615, 2005.

[ASK08a] Ashraf Armoush, Falk Salewski, and Stefan Kowalewski. A hybrid fault tolerance method for recovery block with a weak acceptance test. In *EUC '08: Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 484–491, Washington, DC, USA, 2008. IEEE Computer Society.

[ASK08b] Ashraf Armoush, Falk Salewski, and Stefan Kowalewski. Recovery block with backup voting: A new pattern with extended representation for safety critical embedded systems. In *ICIT '08: Proceedings of the 2008 International Conference on Information Technology*, pages 232–237, Washington, DC, USA, 2008. IEEE Computer Society.

[BBDM02] Davide Bertozzi, Luca Benini, and Giovanni De Micheli. Low power error resilient encoding for on-chip data buses. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 102, Washington, DC, USA, 2002. IEEE Computer Society.

[BBDM05] Davide Bertozzi, Luca Benini, and Giovanni De Micheli. Error control schemes for on-chip communication links: the energy-reliability tradeoff. *TCAD, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(6):818–831, June 2005.

[BC06] Luciano Bononi and Nicola Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 154–159, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[BCGK04] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: Qos architecture and design process for network on chip. *J. Syst. Archit.*, 50(2-3):105–128, 2004.

[BCGK07] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Routing table minimization for irregular mesh NoCs. In *DATE'07 Proceedings of the conference on Design, automation and test in Europe*, pages 942–947, April 2007.

[BCH+08] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Computer Systems*, 24(1):73 – 84, 2008.

[BMPS03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM.

[BPS06] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for OpenMP programs. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 2–13, New York, NY, USA, 2006. ACM.

[BSCM06] Praveen Bhojwani, Rohit Singhal, Gwan Choi, and Rabi Mahapatra. Forward error correction for on-chip interconnection networks. In *UCAS-II: Proceedings of International Workshop on Unique Chips and Systems*, 2006.

[BSKS01] Kaustav Banerjee, Shukri J. Souri, Pawan Kapur, and Krishna C. Saraswat. 3-D ICs: A novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration. In *Proceedings of the IEEE*, pages 602–633, 2001.

[BSS09] A. Bartzas, K. Siozios, and D. Soudris. Topology exploration and buffer sizing for three-dimensional networks-on-chip. In *DATE Workshop on 3D Integration*, 2009.

[BTEF03] John Bainbridge, Will Toms, Doug Edwards, and Steve Furber. Delay-insensitive, point-to-point interconnect using m-of-n codes. In *ASYNC'03 Proceedings. Ninth International Symposium on Asynchronous Circuits and Systems*, pages 132–140, May 2003.

[CA95] C. M. Cunningham and D. R. Avresky. Fault-tolerant adaptive routing for two-dimensional meshes. In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 122, Washington, DC, USA, 1995. IEEE Computer Society.

[CAZN10] Fabien Chaix, Dimiter Avresky, Nacer-Eddine Zergainoh, and Michael Nicolaidis. Fault-tolerant deadlock-free adaptive routing for any set of link and node failures in multi-core systems. In *NCA '10: The 9th IEEE International Symposium on Network Computing and Applications*, 2010. To appear.

[CB97] Suresh Chalasani and Rajendra V. Boppana. Communication in multicomputers with nonconvex faults. *IEEE Trans. Comput.*, 46(5):616–622, 1997.

[CC01] Chun-Lung Chen and Ge-Ming Chiu. A fault-tolerant routing scheme for meshes with nonconvex faults. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):467–475, 2001.

[CC02] Subhachandra Chandra and Peter M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering*, page 91, Washington, DC, USA, 2002. IEEE Computer Society.

[CCCM06] G. Campobello, M. Castano, C. Ciofi, and D. Mangano. GALS networks on chip: a new solution for asynchronous delay-insensitive links. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 160–165, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM.

[CGL+08] Marcello Coppola, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia, and Lorenzo Pieralisi. *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*. CRC Press, Inc., Boca Raton, FL, USA, 2008.

[CH84] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: a state-of-the-art review. *IBM J. Res. Dev.*, 28(2):124–134, 1984.

[CIB09] Nicola Concer, Salvatore Iamundo, and Luciano Bononi. aEqualized: A novel routing algorithm for the spidergon network on chip. In *DATE*, pages 749–754. IEEE, 2009.

[CKS01] Xavier Caron, Jörg Kienzle, and Alfred Strohmeier. Object-oriented stable storage based on mirroring. In *Ada Europe '01: Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, pages 278–289, London, UK, 2001. Springer-Verlag.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CMH96] F. Cristian, S. Mishra, and Y. S. Hyun. Implementation and performance of a stable-storage service in unix. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems*, page 86, Washington, DC, USA, 1996. IEEE Computer Society.

[Con08] Cristian Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *RAMS '08: Proceedings of the 2008 Annual Reliability and Maintainability Symposium*, pages 370–374, Washington, DC, USA, 2008. IEEE Computer Society.

[CS98] Guohong Cao and Mukesh Singhal. On coordinated checkpointing in distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(12):1213–1225, 1998.

[CS01] Guohong Cao and Mukesh Singhal. Mutable checkpoints: A new checkpointing approach for mobile computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):157–172, 2001.

[CS09] Charles Chiang and Subarna Sinha. The road to 3D EDA tool readiness. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 429–436, Piscataway, NJ, USA, 2009. IEEE Press.

[CYR09] Nianen Chen, Yue Yu, and Shangping Ren. Checkpoint interval and system's overall quality for message logging-based rollback and recovery in distributed and embedded computing. In *ICESS '09: Proceedings of the 2009 International Conference on Embedded Software and Systems*, pages 315–322, Washington, DC, USA, 2009. IEEE Computer Society.

[Dal06] W.J. Dally. Future directions for on-chip interconnection networks. In *OCIN Workshop*, 2006.

[dARGG06] David de Andrés, Juan Carlos Ruiz, Daniel Gil, and Pedro J. Gil. Fast emulation of permanent faults in VLSI systems. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*, pages 1–6. IEEE, 2006.

[dCGKG04] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 35–40, New York, NY, USA, 2004. ACM.

[DCR03] Shamik Das, Anantha Chandrakasan, and Rafael Reif. Three-dimensional integrated circuits: Performance, design methodology, and cad tools. *VLSI, IEEE Computer Society Annual Symposium on*, 0:13, 2003.

[DDF+08] G. Druais, G. Dilliway, P. Fischer, E. Guidotti, O. Lühn, A. Radisic, and S. Zahraoui. High aspect ratio via metallization for 3D integration using CVD TiN barrier and electrografted cu seed. *Microelectron. Eng.*, 85(10):1957–1961, 2008.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[DM03] Tudor Dumitras and Radu Marculescu. On-chip stochastic communication. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10790, Washington, DC, USA, 2003. IEEE Computer Society.

[DM05] Giovanni Denaro and Leonardo Mariani. Towards testing and analysis of systems that use serialization. *Electron. Notes Theor. Comput. Sci.*, 116:171–184, 2005.

[DM09] Giovanni De Micheli. An outlook on design technologies for future integrated systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(6):777–790, 2009.

[DS94] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. pages 345–351, 1994.

[DT07] Avijit Dutta and Nur A. Touba. Reliable network-on-chip using a low cost unequal error protection code. In *DFT '07: Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 3–11, Washington, DC, USA, 2007. IEEE Computer Society.

[Dum01] James M. Dumoulin. Apollo-11. 2001. http://science.ksc.nasa.gov/history/apollo/apollo-11/-apollo-11.html.

[DZK08] Chunjie Duan, Chengyu Zhu, and Sunil P. Khatri. Forbidden transition free crosstalk avoidance codec design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 986–991, New York, NY, USA, 2008. ACM.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[EJZ92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *In Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[EP04] Elmootazbellah N. Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secur. Comput.*, 1(2):97–108, 2004.

[FDC+09] David Fick, Andrew DeOrio, Gregory K. Chen, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A highly resilient routing algorithm for fault-tolerant NoCs. In *DATE*, pages 21–26, 2009.

[FKCC06] Arthur Pereira Frantz, Fernanda Lima Kastensmidt, Luigi Carro, and Erika Cota. Dependable network-on-chip router able to simultaneously tolerate soft errors and crosstalk. pages 1 –9, oct. 2006.

[FL06] Thomas Huining Feng and Edward A. Lee. Incremental checkpointing with application to distributed discrete event simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1004–1011. Winter Simulation Conference, 2006.

[FRB08] Cesare Ferri, Sherief Reda, and R. Iris Bahar. Parametric yield management for 3D ics: Models and strategies for improvement. *J. Emerg. Technol. Comput. Syst.*, 4(4):1–22, 2008.

[Fur06] Steve Furber. Living with failure: Lessons from nature? In *ETS '06: Proceedings of the Eleventh IEEE European Test Symposium*, pages 4–8, Washington, DC, USA, 2006. IEEE Computer Society.

[GAP+07] Cristian Grecu, Lorena Anghel, Partha P. Pande, Andre Ivanov, and Resve Saleh. Essential fault-tolerance metrics for NoC infrastructures. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 37–42, Washington, DC, USA, 2007. IEEE Computer Society.

[GHKP07] Qi Gao, Wei Huang, Matthew J. Koop, and Dhabaleswar K. Panda. Group-based coordinated checkpointing for MPI: A case study on infiniband. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 47, Washington, DC, USA, 2007. IEEE Computer Society.

[GIS+06] Cristian Grecu, André Ivanov, Res Saleh, Egor S. Sogomonyan, and Partha Pratim Pande. On-line fault detection and location for NoC interconnects. In *IOLTS'06 Proceedings of IEEE International On-Line Testing Symposium*, July 2006.

[GISP06] Cristian Grecu, Andre Ivanov, Res Saleh, and Partha Pratim Pande. NoC interconnect yield improvement using crosspoint redundancy. In *DFT '06: Proceedings of the 21st IEEE International Symposium on on Defect and Fault-Tolerance in VLSI Systems*, pages 457–465, Washington, DC, USA, 2006. IEEE Computer Society.

[GMRV06] Bidyut Gupta, Namdar Mogharreban, Shahram Rahimi, and A. Vemuri. A high performance non-blocking checkpointing/recovery algorithm for ring networks. In *PDPTA*, pages 234–240, 2006.

[GPBG08] Amlan Ganguly, Partha Pratim Pande, Benjamin Belzer, and Cristian Grecu. Design of low power & reliable networks on chip through joint crosstalk avoidance and multiple error correction coding. *J. Electron. Test.*, 24(1-3):67–81, 2008.

[GR06] Bidyut Gupta and Shahram Rahimi. A fast and efficient non-blocking coordinated checkpointing approach for distributed systems. In *PDPTA*, pages 99–105, 2006.

[GR09] B. Gupta and S. Rahimi. A novel low-overhead recovery approach for distributed systems. *J. Comp. Sys., Netw., and Comm.*, 2009:1–8, 2009.

[GYHP06] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for MPI programs over infiniband. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[HB05] Min Huang and Brett Bode. A performance comparison of tree and ring topologies in distributed systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 13*, page 258.1, Washington, DC, USA, 2005. IEEE Computer Society.

[HCG07] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 954–959, San Jose, CA, USA, 2007. EDA Consortium.

[HGR05] Andreas Hansson, Kees Goossens, and Andrei Radulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2005. ACM.

[HJS04] David Hodges, Horace Jackson, and Resve Saleh. *Analysis and Design of Digital Integrated Circuits*. McGraw-Hill, Inc., New York, NY, USA, 2004.

[HKC+01] Jiman Hong, Sangsu Kim, Yookun Cho, H. Y. Yeom, and T aesoon Park. On the choice of checkpoint interval using memory usage profile and adaptive time series analysis. In *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[HM04] Jingcao Hu and Radu Marculescu. Dyad: smart routing for networks-on-chip. In *DAC'04 Proceedings. 41st Design Automation Conference*, 2004.

[HS04] Ching-Tien Ho and Larry Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. *IEEE Trans. Comput.*, 53(4):427–439, 2004.

[Hsi70] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM J. Res. Dev.*, 14(4):395–401, 1970.

[IFI] IFIP. International Federation for Information Processing. http://www.dependability.org/wg10.4/.

[Int05] Intel. Excerpts from A Conversation with Gordon Moore: Moore's Law. 2005. ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.

[ITR05] ITRS. International Technology Roadmap for Semiconductors. In *Interconnect*, 2005. http://www.itrs.net.

[ITR07] ITRS. International Technology Roadmap for Semiconductors. In *Interconnect*, 2007. http://www.itrs.net.

[ITR09a] ITRS. International Technology Roadmap for Semiconductors. In *System Drivers*, 2009. http://www.itrs.net.

[ITR09b] ITRS. International Technology Roadmap for Semiconductors. In *Process Integration, Devices, and Structures*, 2009. http://www.itrs.net.

[ITR09c] ITRS. International Technology Roadmap for Semiconductors. In *Design*, 2009. http://www.itrs.net.

[JKG09] Samir Jafar, Axel Krings, and Thierry Gautier. Flexible rollback recovery in dynamic heterogeneous grid computing. *IEEE Trans. Dependable Secur. Comput.*, 6(1):32–44, 2009.

[JLV05] Axel Jantsch, Robert Lauter, and Arseni Vitkovski. Power analysis of link level and end-to-end data protection in networks on chip. In *ISCAS (2)*, pages 1770–1773, 2005.

[JS04] Amit Jain and R. K. Shyamasundar. Failure detection and membership management in grid environments. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 44–52, Washington, DC, USA, 2004. IEEE Computer Society.

[KCR06] Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[KNM04] Claudia Kretzschmar, André K. Nieuwland, and Dietmar Müller. Why transition coding for power minimization of on-chip buses does not work. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10512, Washington, DC, USA, 2004. IEEE Computer Society.

[KNP+06] Jongman Kim, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Mazin S. Yousif, and Chita R. Das. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. In *ISCA'06 33rd International Symposium on Computer Architecture*, pages 4–15, 2006.

[KT86] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1):23–31, 1987.

[KV01] Ralf Koetter and Alexander Vardy. Algebraic soft-decision decoding of reed-solomon codes. *IEEE Trans. Inform. Theory*, 49:2809–2825, 2001.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LBKC04] Pierre Lemarinier, Aurelien Bouteiller, Geraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *Int. J. High Perform. Comput. Netw.*, 2(2-4):146–155, 2004.

[Led07] Patrick Leduc. 3D integration: A solution for interconnects. Technical report, 2007.

[Li08] Jian Li. 3D integration opportunities and challenges. In *RFID '07: IEEE International Conference on Radio Frequency Identification*, pages 175–182. ISCA Tutorial, 2008.

[LL08] Jean-Rene Lequepeys and Didier Lattard. Trends in complex SoC design: From technology variability to multiprocessor architectures. In *FP7 Workshop Design*, 2008.

[LLP07] Teijo Lehtonen, Pasi Liljeberg, and Juha Plosila. Online reconfigurable self-timed links for fault tolerant NoC. *VLSI Design*, 2007:13, 2007.

[LLP09] T. Lehtonen, P. Liljeberg, and J. Plosila. Fault tolerant distributed routing algorithms for mesh networks-on-chip. In *ISSCS '09: International Symposium on Signals, Circuits and Systems*, pages 1–4, 2009.

[LML+08] Igor Loi, Subhasish Mitra, Thomas H. Lee, Shinobu Fujita, and Luca Benini. A low-overhead fault tolerance scheme for TSV-based 3D network on chip links. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 598–602, Piscataway, NJ, USA, 2008. IEEE Press.

[LMS+05] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest. Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 81–86, New York, NY, USA, 2005. ACM.

[LPC05] Yamin Li, Shietung Peng, and Wanming Chu. Adaptive box-based efficient fault-tolerant routing in 3D torus. pages 71–77, 2005.

[LSZ+09] Yinghai Lu, Li Shang, Hai Zhou, Hengliang Zhu, Fan Yang, and Xuan Zeng. Statistical reliability analysis under process variation and aging effects. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 514–519, New York, NY, USA, 2009. ACM.

[LTW07] Nur A. Touba Laung-Terng Wang, Charles E. Stroud. *System-on-Chip Test Architectures*. Morgan Kaufmann, 2007.

[LVKI03] Lin Li, N. Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Adaptive error protection for energy efficiency. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 2, Washington, DC, USA, 2003. IEEE Computer Society.

[LYMC07] Zhi Li, Lihua Yuan, Prasant Mohapatra, and Chen-Nee Chuah. On the analysis of overlay failure detection and recovery. *Comput. Netw.*, 51(13):3828–3843, 2007.

[MABDM06] Srinivasan Murali, David Atienza, Luca Benini, and Giovanni De Micheli. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 845–848, New York, NY, USA, 2006. ACM.

[MBT+05] Srinivasan Murali, Luca Benini, Theocharis Theocharides, N. Vijaykrishnan, Mary Jane Irwin, and Giovanni De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design and Test of Computers*, 22(5):434–442, 9-10 2005.

[MCM+05] Cesar Marcon, Ney Calazans, Fernando Moraes, Altamiro Susin, Igor Reis, and Fabiano Hessel. Exploring NoC mapping strategies: An energy and timing aware technique. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 502–507, Washington, DC, USA, 2005. IEEE Computer Society.

[MFVP08] Neeraj Mittal, Felix C. Freiling, S. Venkatesan, and Lucia Draque Penso. On termination detection in crash-prone distributed systems with failure detectors. *J. Parallel Distrib. Comput.*, 68(6):855–875, 2008.

[MKWS04] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *SLIP '04: Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13, New York, NY, USA, 2004. ACM.

[MM04] Partha Sarathi Mandal and Krishnendu Mukhopadhyaya. Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks. *J. Parallel Distrib. Comput.*, 64(5):649–661, 2004.

[MMB04] Sebastien Monnet, Christine Morin, and Ramamurthy Badrinath. A hierarchical checkpointing protocol for parallel applications in cluster federations. *Parallel and Distributed Processing Symposium, International*, 12:211a, 2004.

[Moo05] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[MRR97] F. Moll, M. Roca, and A. Rubio. Measurement of crosstalk-induced delay errors in integrated circuits. *Electronics Letters*, 33(19):1623–1624, 1997.

[Mut07] Madhu Mutyam. Selective shielding: a crosstalk-free bus encoding technique. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 618–621, Piscataway, NJ, USA, 2007. IEEE Press.

[NGF+04] N A Nordbotten, M E Gómez, J Flich, P Lopez, A Robles, T Skeie, O Lysne, and J Duato. A fully adaptive fault-tolerant routing methodology based on intermediate nodes. pages 341–356, 2004.

[Nic05] Michael Nicolaidis. Design for soft error mitigation. *IEEE Trans. Device and Matl. Reliability*, 5(3):405–418, 2005.

[NKRM05] Andre K. Nieuwland, Atul Katoch, Daniele Rossi, and Cecilia Metra. Coding techniques for low switching noise in fault tolerant busses. In *IOLTS '05: Proceedings of the 11th IEEE International On-Line Testing Symposium*, pages 183–189, Washington, DC, USA, 2005. IEEE Computer Society.

[Par96] B. Parhami. Design of reliable software via general combination of n-version programming and acceptance testing. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, page 104, Washington, DC, USA, 1996. IEEE Computer Society.

[Pas09] Sudeep Pasricha. Exploring serial vertical interconnects for 3D ICs. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 581–586, New York, NY, USA, 2009. ACM.

[PC05] Ishwar Parulkar and Robert Cypher. Trends and trade-offs in designing highly robust throughput computing oriented chips and systems. In *IOLTS '05: Proceedings of the 11th IEEE International On-Line Testing Symposium*, pages 74–77, Washington, DC, USA, 2005. IEEE Computer Society.

[PGC06] Jay A. Patel, Indranil Gupta, and Noshir Contractor. Jetstream: Achieving predictable gossip dissemination by leveraging social network principles. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 32–39, Washington, DC, USA, 2006. IEEE Computer Society.

[PGK00] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks(raid). pages 474–481, 2000.

[PHKC09] Maurizio Palesi, Rickard Holsmark, Shashi Kumar, and Vincenzo Catania. Application specific routing algorithms for networks on chip. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):316–330, 2009.

[PKH06] Maurizio Palesi, Shashi Kumar, and Rickard Holsmark. A method for router table compression for application specific routing in mesh topology NoC architectures. In *Proc. International Workshop on Architectures, Modeling, and Simulation*, July 2006.

[PLB+04] Matthew Pirretti, Greg M. Link, Richard R. Brooks, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Fault tolerant algorithms for network-on-chip interconnect. In *VLSI'04 Proceedings. IEEE Computer society Annual Symposium on VLSI*, 2004.

[PM04] Ketan N. Patel and Igor L. Markov. Error-correction and crosstalk avoidance in DSM busses. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(10):1076–1080, 2004.

[PNK+06] Dongkook Park, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, and Chita R. Das. Exploring fault-tolerant network-on-chip architectures. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 93–104, Washington, DC, USA, 2006. IEEE Computer Society.

[PZGG06] Partha Pratim Pande, Haibo Zhu, Amlan Ganguly, and Cristian Grecu. Energy reduction through crosstalk avoidance coding in NoC paradigm. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 689–695, Washington, DC, USA, 2006. IEEE Computer Society.

[QS03] Francesco Quaglia and Andrea Santoro. Modeling and optimization of non-blocking checkpointing for optimistic simulation on myrinet clusters. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 130–139, New York, NY, USA, 2003. ACM.

[Rac09] Harald Racke. Survey on oblivious routing strategies. In *CiE '09: Proceedings of the 5th Conference on Computability in Europe*, pages 419–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[RAM07] Daniele Rossi, Paolo Angelini, and Cecilia Metra. Configurable error control scheme for NoC signal integrity. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 43–48, Washington, DC, USA, 2007. IEEE Computer Society.

[Ran75] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[RGR+03] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10350, Washington, DC, USA, 2003. IEEE Computer Society.

[RNKM05] Daniele Rossi, Andre K. Nieuwland, Atul Katoch, and Cecilia Metra. Exploiting ecc redundancy to minimize crosstalk impact. *IEEE Des. Test*, 22(1):59–70, 2005.

[RS60] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[RS97] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogeneous archtitectures. page 58, 1997.

[SABR05] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Lifetime reliability: Toward an architectural solution. *IEEE Micro*, 25(3):70–80, 2005.

[Sca07] Mark Scannell. CEA-Leti 3D activities and roadmap. In *EMC-3D European Technical Symposium*, 2007.

[Sch84] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[SFJ03] Kuo-Feng Ssu, W. Kent Fuchs, and Hewijin C. Jiau. Process recovery in heterogeneous systems. *IEEE Trans. Comput.*, 52(2):126–138, 2003.

[SK07] Toshinori Sato and Yuji Kunitake. A simple flip-flop circuit for typical-case designs for DFM. In *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*, pages 539–544, Washington, DC, USA, 2007. IEEE Computer Society.

[SMBM09] Ciprian Seiculescu, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Sunfloor 3D: A tool for networks on chip topology synthesis for 3D systems on chips. In *DATE '09: Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 9–14. IEEE, 2009.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems, 1983.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[SS04] Srinivasa R. Sridhara and Naresh R. Shanbhag. Coding for system-on-chip networks: a unified framework. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 103–106, New York, NY, USA, 2004. ACM.

[SSC96] L. M. Silva, J. G. Silva, and S. Chapple. Portable transparent checkpointing for distributed shared memory. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 422, Washington, DC, USA, 1996. IEEE Computer Society.

[ST07] Falk Salewski and Adam Taylor. Fault handling in FPGAs and microcontrollers in safety-critical embedded applications: A comparative survey. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 124–131, Washington, DC, USA, 2007. IEEE Computer Society.

[TDB03] Brian Towles, William J. Dally, and Stephen Boyd. Throughput-centric routing algorithm design. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 200–209, 2003.

[Tez08] Tezzaron Semiconductor. 3D-ICs and integrated circuit security, 2008. http://www.tezzaron.com/about/papers/3D-ICs_and_Integrated_Circuit_Security.pdf.

[TM09] Dietmar Tutsch and Miroslaw Malek. Comparison of network-on-chip topologies for multicore systems considering multicast and local traffic. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–9, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[VBC05] Praveen Vellanki, Nilanjan Banerjee, and Karam S. Chatha. Quality-of-service and error control techniques for mesh-based network-on-chip architectures. *Integr. VLSI J.*, 38(3):353–382, 2005.

[Wil00] Torres Wilfredo. Software fault tolerance: A tutorial. Technical report, 2000.

[WL06] Eric Wong and Sung Kyu Lim. 3D floorplanning with thermal vias. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 878–883, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[Wu03] Jie Wu. A simple fault-tolerant adaptive and minimal routing approach in 3-D meshes. *J. Comput. Sci. Technol.*, 18(1):1–13, 2003.

[XSW05] Dong Xiang, Jia-Guang Sun, and Jie Wu. Fault-tolerant routing in meshes/tori using planarly constructed fault blocks. pages 577–584, 2005.

[YA08] Qiaoyan Yu and Paul Ampadu. Adaptive error control for NoC switch-to-switch links in a variable noise environment. In *DFT '08: Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 352–360, Washington, DC, USA, 2008. IEEE Computer Society.

[YW99] Yuanyuan Yang and Jianchao Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 290, Washington, DC, USA, 1999. IEEE Computer Society.

[ZGT08] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for a fault-tolerant 2D-mesh network-on-chip. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 441–446, New York, NY, USA, 2008. ACM.

[Zim88] H. Zimmermann. OSI reference model—the ISO model of architecture for open systems interconnection. pages 2–9, 1988.

[ZJ03] Heiko Zimmer and Axel Jantsch. A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 188–193, New York, NY, USA, 2003. ACM.

[ZLKK07] Yizheng Zhou, Vijay Lakamraju, Israel Koren, and C. M. Krishna. Software-based failure detection and recovery in programmable network interfaces. *IEEE Trans. Parallel Distrib. Syst.*, 18(11):1539–1550, 2007.

[ZP06] Kun Zhang and Santosh Pande. Minimizing downtime in seamless migrations of mobile applications. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 12–21, New York, NY, USA, 2006. ACM.

Publications

- Claudia Rusu, Lorena Anghel, Dimiter Avresky, "Adaptive Inter-Layer Message Routing in 3D Networks-on-Chip", Microprocessors and Microsystems Journal 2010. Accepted.
- Claudia Rusu, Lorena Anghel, Dimiter Avresky, "RILM: Reconfigurable inter-layer routing mechanism for 3D multi-layer networks-on-chip", 16th IEEE International On-Line Testing Symposium (IOLTS), 2010, Corfu Island, Greece.
- Vladimir Pasca, Lorena Anghel, Claudia Rusu, Mounir Benabdenbi, "Configurable serial fault-tolerant link for communication in 3D integrated systems", 16th IEEE International On-Line Testing Symposium (IOLTS), 2010, Corfu Island, Greece.
- Vladimir Pasca, Lorena Anghel, Claudia Rusu, Mounir Benabdenbi, "Configurable fault-tolerant link for inter-die communication in 3D on-chip networks", 15th IEEE European Test Symposium (ETS), 2010, Prague, Czech Republic.
- Vladimir Pasca, Lorena Anghel, Claudia Rusu, Mounir Benabdenbi, "Non-regular 3D mesh networks-on-chip", DAC Workshop on Diagnostic Services in Network-on-Chips (DSNoC), 2010, Anaheim, California, USA.
- Claudia Rusu, Lorena Anghel, "Checkpoint and rollback recovery in network-on-chip based systems", Student forum at ASP-DAC 2010, Taipei, Taiwan.
- Vladimir Pasca, Lorena Anghel, Claudia Rusu, Ricardo Locatelli, Marcello Coppola, "Error resilience of intra-die and inter-die communication with 3D Spidergon STNoC", DATE, 2010, Dresden, Germany.
- Claudia Rusu, Lorena Anghel, Dimiter Avresky, "Message routing in 3D networks-on-chip", NORCHIP, 2009, Trondheim, Norway.
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Efficient coordinated checkpointing recovery schemes for network-on-chip based systems", 2nd International Workshop on Dependable Circuit Design (DECIDE), 2008, Playa del Carmen, Mexico.
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Network-on-chip fault tolerance through checkpoint and rollback recovery", National Symposium on System-on-Chip – System-in-Package (GdR SoC-SiP), 2008, Paris, France
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Communication-aware recovery configurations for networks-on-chip", 14th IEEE International On-Line Testing Symposium (IOLTS), 2008, Rhodes, Greece.
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Blocking and non-blocking checkpointing for networks-on-chip", 2nd IEEE Workshop on Dependable and Secure Nanocomputing (WDSN), 2008, Anchorage, Alaska, USA.
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Improving the scalability of checkpoint recovery for networks-on-chip", IEEE International Symposium on Circuits and Systems (ISCAS), 2008, Seattle, Washington, USA.
- Claudia Rusu, Cristian Grecu, Lorena Anghel, "Coordinated versus uncoordinated checkpoint recovery for network-on-chip based systems", 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA), 2008, Hong Kong.
- Cristian Grecu, André Ivanov, Resve Saleh, Claudia Rusu, Lorena Anghel, Partha P. Pande, Vasile Nuca, **"**A flexible network-on-chip simulator for early design space exploration", 1st Microsystems and Nanoelectronics Research Conference (MNRC), 2008, Ottawa, Canada.
- Claudia Rusu, Antonin Bougerol, Lorena Anghel, C. Weulerse, Nadine Buard, Seddik Benhammadi, Nicolas Renaud, Guillaume Hubert, Frédéric Wrobel, T. Carriere, R. Gaillard, "Multiple event transient induced by nuclear reactions in CMOS logic cells", 13th IEEE International On-Line Testing Symposium (IOLTS), 2007, Heraklion, Greece.

Tolérance aux fautes multi-niveau dans les réseaux sur puce

Résumé – Avec la diminution continue des caractéristiques technologiques et la complexité croissante des systèmes sur puce, les réseaux sur puce se sont imposés comme la solution la plus prometteuse pour assurer la communication entre les différents composants intégrés. Toutefois, les systèmes sur puce actuels sont soumis à différents facteurs perturbants (variation du processus, électromigration, interférences, l'environnement radiatif et des défauts permanents dans le cas de l'intégration 3D). Ces facteurs peuvent perturber le bon fonctionnement logique et temporel, et conduire aux défaillances du système de communication ou des différentes entités d'un système hétérogène à base de multipro-cesseurs, ASICs, mémoires etc.

Dans cette these, on s'intéresse aux différentes approches complémentaires pour améliorer la tolerance aux fautes des réseaux sur puce. Cette complementarité s'applique sur les differentes couches des protocoles de communication : détection d'erreurs et correction ou retransmission de la donnée au niveau de la couche de liaison, algorithmes de routage tolérants aux fautes pour les topologies 3D pour la couche réseau, et solutions de recouvrement de défaillances en utilisant les points de contrôle pour la couche application.

Mots clés : réseau sur puce, intégration 3D, tolérance aux fautes, recouvrement par points de contrôle, routage, reconfiguration, détection d'erreurs, correction d'erreurs

---

Multi-Level Fault-Tolerance in Networks-on-Chip

Abstract – With the continuous shrinking of technology features and the growing complexity of systems-on-chip, networks-on-chip have emerged as the most promising solution for the on-chip communication system. However, current systems-on-chip are subject to different factors (process variation, electromigration, crosstalk, environmental constraints and permanent defects in the case of 3D integration) that can perturb their logical and temporal operation and eventually lead to failures of the communication system or of different components of heterogeneous systems comprised of microprocessors, ASICs, memories etc. In this thesis different complementary approaches to deal with these problems are addressed, including techniques at data link level such as error detection combined with correction or data retransmission, fault tolerant routing algorithms for 3D topologies, and rollback recovery of the application after possible failures, by the use of checkpoints.

Keywords: network-on-chip, 3D integration, fault-tolerance, rollback recovery, checkpoint, routing, reconfiguration, error detection, error correction

---

Adresse : Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France
ISBN : 978-2-84813-158-0