PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LAYERED APPROACH FOR RUNTIME FAULT RECOVERY IN NOC-BASED MPSOCS

EDUARDO WEBER WÄCHTER

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: Prof. Dr. Fernando Gehm Moraes

Co-Advisor: Prof. Dr. Alexandre de Morais Amory

Porto Alegre, Brasil
2015

Dados Internacionais de Catalogação na Publicação (CIP)

W114l Wächter, Eduardo Weber
Layered approach for runtime fault recovery in NOC-Based MPSOCS / Eduardo Weber Wächter. – Porto Alegre, 2015.
91 p.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fernando Gehm Moraes
Co-orientador: Prof. Dr. Alexandre de Morais Amory

1. Informática. 2. Arquitetura de Computador.
3. Microprocessadores. I. Moraes, Fernando Gehm. II. Amory, Alexandre de Morais. III. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "*Layered Approach for Runtime Fault Recovery in NoC-based MPSoCs*", apresentada por Eduardo Weber Wächter, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, aprovada em 10/06/2015 pela Comissão Examinadora:

Prof. Dr. Fernando Gehm Moraes — PPGCC/PUCRS
Orientador

Prof. Dr. Alexandre de Morais Amory — PPGCC/PUCRS
Coorientador

Prof. Dr. César Augusto Missio Marcon — PPGCC/PUCRS

Profa. Dra. Fernanda Gusmão de Lima Kastensmidt — UFRGS

Profa. Dra. Letícia Maria Bolzani Pöhls — PPGEE/PUCRS

Prof. Dr. Nicolas Ventroux — CEA LIST

Homologada em 21/08/2015, conforme Ata No. 015 pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

O doutorado é um caminho, longo diga-se de passagem, que não é feito sozinho. São quatro longos anos de muito trabalho, preocupações e cabelos brancos. Dessa forma primeiramente agradecer aos meus pais, meus maiores incentivadores. Sei que independentemente do que aconteça, é pra lá que a gente vai quando a coisa aperta. Eles me apoiaram muito neste caminho. Amo vocês.

Sempre digo que sem meu orientador essa tese não teria saído. Meu enésimo muito obrigado ao meu orientador Fernando Moraes, exemplo de pesquisador e professor. Obrigado pelo tempo que dedicou na tese, nas reuniões, nos brainstorms.

Ao meu co-orientador que conheci no final do mestrado, Alexandre Amory, por trazer mais "brasa ao assado". Essa tese tem muita contribuição tua. Muito Obrigado.

Ao apoio financeiro da CAPES/Fapergs, que além da bolsa me proporcionaram uma experiência única no exterior. Essa experiência me mostrou quais são as prioridades na vida.

Ao Nicolas Ventroux, meu orientador na França e ao chefe do LCE Raphäel David por me receberem muito bem e me aceitarem no grupo. Serei eternamente grato.

Aos amigos que fiz e que mantenho no laboratório, obrigado pelas boas gargalhadas e cervejas! Valeu Mandelli, Matheus, Castilhos, Guindani, Raupp, Julian, Ost, Leonel, Edson, Leonardo, Augusto, Carlos Henrique, Walter, Ruaro, Felipe, Heck, Sérgio, Thiago, Madalozzo.

Aos integrantes e amigos do FBC e da Xorna, o meu muito obrigado por me tirar da rotina quando eu precisava, e algumas vezes quando eu não precisava também!

Por fim, gostaria de agradecer ao amor da minha vida. A pessoa que faz os meus dias serem mais felizes, Keli, te amo. Obrigado por dizer que tudo ia dar certo, quando não sabia se ia dar certo. Obrigado por acreditar em mim.

ABORDAGEM EM CAMADAS PARA RECUPERAÇÃO DE FALHAS EM TEMPO DE EXECUÇÃO EM MPSOCS BASEADOS EM NOC

RESUMO

Mecanismos de tolerância a falhas em MPSoCs são obrigatórios para enfrentar defeitos ocorridos durante a fabricação ou falhas durante a vida útil do circuito integrado. Por exemplo, falhas permanentes na rede de interconexão do MPSoC podem interromper aplicações mesmo que a rede tenha caminhos sem falha para um determinado destino. A tolerância a falhas em tempo de execução fornece mecanismos de auto-organização para continuar a oferecer serviços de processamento apesar de núcleos defeituosos devido à presença de falhas permanentes e/ou transitórias durante toda a vida dos chips. Esta Tese apresenta uma abordagem em camadas para um MPSoC tolerante a falhas, onde cada camada é responsável por resolver uma parte do problema. O método é construído sobre uma nova proposta de rede especializada utilizada para procurar caminhos livre de falha. A primeira camada, denominada camada física, é responsável pela detecção de falhas e isolamento das partes defeituosas da rede. A segunda camada, denominada camada de rede, é responsável por substituir um caminho defeituoso por um caminho alternativo livre de falhas. Um método de roteamento tolerante a falhas executa o mecanismo de busca de caminhos e reconfigura a rede para usar este caminho livre de falhas. A terceira camada, denominada camada de transporte, implementa um protocolo de comunicação tolerante a falhas que detecta quando pacotes não são entregues ao destino, acionando o método proposto na camada de rede. A última camada, camada de aplicação, é responsável por mover as tarefas do elemento de processamento (PE) defeituoso para um PE saudável, salvar o estado interno da tarefa, e restaurá-la em caso de falha durante a execução. Os resultados na camada de rede mostram um método rápido para encontrar caminhos livres de falhas. O processo de procura de caminhos alternativos leva tipicamente menos de 2000 ciclos de relógio (ou 20 microssegundos). Na camada de transporte, diferentes abordagens foram avaliadas para detectar uma mensagem não entregue e acionar a retransmissão. Os resultados mostram que a sobrecarga para retransmitir a mensagem é 2,46 vezes maior quando comparado com o tempo para transmitir uma mensagem sem falha, sendo que todas outras mensagens subsequentes são transmitidas sem sobrecarga. Para as aplicações DTW, MPEG e sintética, o caso médio de sobrecarga no tempo de execução da aplicação é de 0,17%, 0,09% e 0,42%, respectivamente. Isto representa menos do que 5% do tempo de execução de uma dada aplicação no pior caso. Na camada de aplicação, todo o protocolo de recuperação de falhas executa rapidamente, com uma baixa sobrecarga no tempo de execução sem falhas (5,67%) e com falhas (17,33% - 28,34%).

Palavras Chave: Sistemas Multi-Processados em Chip (MPSoC), Redes Intra-chip (NoC), Tolerância a Falhas.

LAYERED APPROACH FOR RUNTIME FAULT RECOVERY IN NOC-BASED MPSOCS

ABSTRACT

Mechanisms for fault-tolerance in MPSoCs are mandatory to cope with defects during fabrication or faults during product lifetime. For instance, permanent faults on the interconnect network can stall or crash applications, even though the MPSoCs' network has alternative fault-free paths to a given destination. Runtime Fault Tolerance provide self-organization mechanisms to continue delivering their processing services despite defective cores due to the presence of permanent and/or transient faults throughout their lifetime. This Thesis presents a runtime layered approach to a fault-tolerant MPSoC, where each layer is responsible for solving one part of the problem. The approach is built on top of a novel small specialized network used to search fault-free paths. The first layer, named physical layer, is responsible for the fault detection and fault isolation of defective routers. The second layer, named the network layer, is responsible for replacing the original faulty path by an alternative fault-free path. A fault-tolerant routing method executes a path search mechanism and reconfigures the network to use the faulty-free path. The third layer, named transport layer, implements a fault-tolerant communication protocol that triggers the path search in the network layer when a packet does not reach its destination. The last layer, application layer, is responsible for moving tasks from the defective processing element (PE) to a healthy PE, saving the task's internal state, and restoring it in case of fault while executing a task. Results at the *network layer*, show a fast path finding method. The entire process of finding alternative paths takes typically less than 2000 clock cycles or 20 microseconds. In the *transport layer*, different approaches were evaluated being capable of detecting a lost message and start the retransmission. The results show that the overhead to retransmit the message is 2.46X compared to the time to transmit a message without fault, being all other messages transmitted with no overhead. For the DTW, MPEG, and synthetic applications the average-case application execution overhead was 0.17%, 0.09%, and 0.42%, respectively. This represents less than 5% of the application execution overhead worst case. At the *application layer*, the entire fault recovery protocol executes fast, with a low execution time overhead with no faults (5.67%) and with faults (17.33% - 28.34%).

Keywords: Multiprocessor System-on-Chip (MPSoC), Network-on-Chip (NoC), Fault Tolerance.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BIST	Built-in self-test
CRC	Cyclic redundancy check
CMOS	Complementary metal-oxide semiconductor
DTW	Dynamic Time Warping
ECC	Error Control Coding
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field Programmable Gate Arrays
FT	Fault-Tolerant
GALS	Globally Asynchronous Locally Synchronous
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HeMPS	Hermes Multiprocessor System
ILP	Instruction-Level Parallelism
IP	Intellectual Property
LUT	Look up Table
MIPS	Microprocessor without Interlocked Pipeline Stages
MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
MPSoC	Multi-Processor System-on-Chip
NI	Network Interface
NoC	Network-on-Chip
OS	Operating System
OSI	Open Systems Interconnection model
PE	Processing Element
SEU	Single Event Upset
SoC	System-on-Chip
TFAP	Time to Find Alternative Paths
TMR	Triple Module Redundancy
TCDM	Tightly Coupled Data Memory
UCD	Unresponsive Communication Detection
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very-Large-Scale integration

SUMMARY

1 INTRODUCTION

From many years to now, processors evolved from single cores with thousands of transistors to many cores with billions of these devices. This evolution enabled the aggressive growth in the number of transistors and the integration of complete systems into a single die. These systems, named Systems-on-Chip (SoCs), are widely used in electronic devices, as in multimedia and telecommunication applications, where high performance and power consumption are important design constraints.

In the past decades, the increase of performance was simply achieved by increasing the frequency of the SoC. However, today a single processor system may not provide the required performance for embedded applications [JER05]. Several “walls” explain this limitation:

- ILP (Instruction Level Parallelism) wall, even with aggressive speculative techniques and large superscalar datapaths, the number of simultaneous instructions executed per clock cycle reached a limit;
- Frequency and power walls, the power increases linearly with the frequency. Power dissipation limits the SoC frequency. Even if CMOS technologies enable operating frequencies superior to 5 GHz, SoCs work at much lower frequencies due to power dissipation issues.

As Figure 1 shows, even if the number of transistors integrated on a single die grows according to Moore’s law, the frequency and power do not follow this trend.

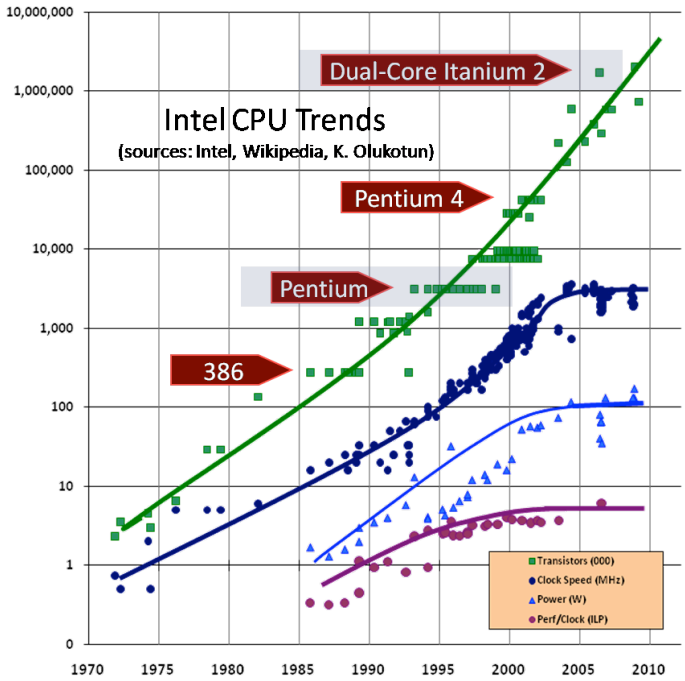


Figure 1 – Evolution of transistor number, clock frequency, power consumption and performance. Adapted from [SUT13].

The alternative to these design and performance gaps is to include more processors in the same die. As a result, instead of using a single processor with a high frequency, new architectures with many simpler and slower processors are used to fill those gaps. These architectures are named Multiprocessor SoCs (MPSoCs). The MPSoC approach raises the performance using a coarse grain parallelism, extending the quest for higher ILP by parallel tasks executing on multiple processors.

The MPSoC trend is nowadays well established, being adopted in GPP (general-purpose processors) and GPU (graphics processing unit) markets. There are many examples of such products: PlayStation 3, which uses the Cell processor [IBM13], with 9 PEs (Processing Elements); Oracle Sparc T5 [FEE13], with 16 PEs. Other examples of MPSoCs include TilePro64 [TIL13] (up to 72 PEs), CSX700 [CLE13] (192 PEs), Teraflops [INT13a] (80 PEs), NVIDIA's Kepler TM GK110 [NVI13] (2,880 PEs) and Single-Chip Cloud Computer [INT13b] (48 PEs). The perspective is that hundreds of PEs in a single die will become mainstream at the end of the decade [ITR15].

As the number of PEs increases, the challenge is to interconnect such PEs in an efficient and scalable fashion. As can be observed in Table 1, the interconnection of large MPSoCs shifts to NoCs (networks-on-chip) [BOR07]. On the other hand, most of the GPUs are an STMD (Single Thread Multiple Data) architecture where all threads execute the same code. For this reason, general-purpose systems adopt message passing (NoCs) and specialized systems, as GPUs, adopt shared memory (hierarchical busses).

The interconnection architectures have evolved from point-to-point links to single and multiple hierarchical buses, and to Networks-on-Chip (NoC) [BEN02]. NoCs are composed of cores connected to routers, and routers interconnected by communication channels [BEN02]. The use of NoCs as the communication infrastructure in these platforms is a reality in academic research and industrial projects, considered an alternative to traditional buses, offering as major advantages scalability and support to multiple parallel communications. However, the motivation for their use in SoCs goes beyond these advantages. NoCs can support many communication services with different levels of quality of service, and offer intrinsically fault tolerance support (more than one path for a source-target pair). The present work assumes general-purpose systems, and hence, NoCs are adopted as the communication infrastructure.

Table 1 – Evolution of number of PEs and interconnection.

Chip	# of PEs	Year	Interconnection
Cell	9	2006	Ring
Teraflops	80	2007	NoC
TilePro64	64	2008	NoC
Single-Chip Cloud Computer	48	2009	NoC
KeplerTM GK110	2880	2012	Hierarchical bus
Oracle Sparc T5	16	2013	Crossbar
CSX700	192	2013	NoC

It is important to differentiate HPC (High-Performance Computing) from embedded systems. In HPC, performance is the main cost function, not power dissipation and energy consumption. An off-chip network interconnects thousands of high-end processors, providing high-performance results. The problem is that, depending on the number of processors, you might need an electrical substation just to deliver the power required by the high-end processors. On the other hand, embedded systems have different constraints, as reduced power consumption, small size, and low cost-per-unit. The present work assumes the design of embedded systems, where such constraints must be fulfilled, together with the performance required by the workload of current embedded applications, like multimedia, telecommunication protocols (as LTE), GPS, augmented reality [ASA06].

In one hand, a larger number of devices in a single die enable the design of MPSoCs, as previously mentioned. On the other hand, the integration of many PEs in a single die reduces the reliability of the system since the probability of faults is related to the transistor channel length and the area of the integrated circuit for a given technology node. In addition, the failure probability is getting higher due to technology miniaturization (Figure 2). Note that same color lines represent failure probability for 1, 5 and 10 years estimation. The combination of the following factors increases the failure rate: (*i*) higher transistors integration; (*ii*) lower voltage levels; (*iii*) higher operation frequency. In highly scaled technology, a variety of wear-out mechanisms can cause transistor failures. As transistor dimensions are getting smaller, effects like oxide breakdown become a concern since the gate oxide tends to become less effective over time. Moreover, negative bias temperature instability [ALA03] is of special concern in PMOS devices, where there is increased threshold voltage over the time. Additionally, thin wires are susceptible to electromigration [GHA82] because conductor material is gradually worn away during chip operation until an open circuit occurs. Since these mechanisms occur over the time, traditional burn-in procedures and manufacturing tests are ineffective to detect them.

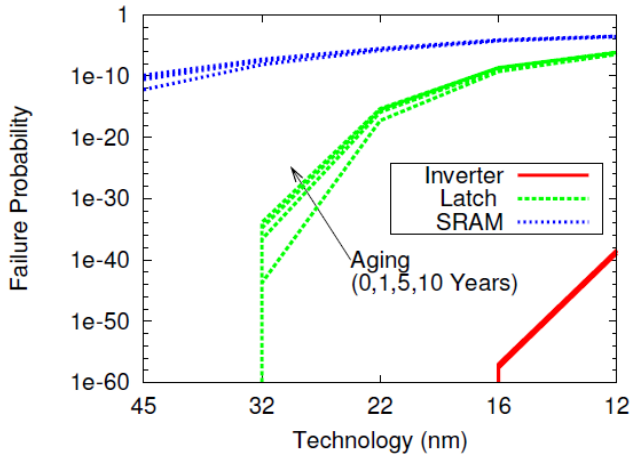


Figure 2 – Failure probability trends as a function of the technology node [SAN10].

All these factors threaten the reliability of future computational systems. Therefore, there is a need to adopt fault-tolerant techniques in the MPSoC design, to maintain the system operation even in the presence of faults. Fault Tolerance (FT) can be defined as the property that enables a system to continue to operate correctly in the presence of a failure in some of its components. Section 1.1 describes an example of how a single fault in a single processor of a MPSoC can compromise the whole system.

NoC-based MPSoCs provide excellent opportunities for the design of reliable systems. For example, there are redundant paths between processors and it is possible to design an integrated hardware-software fault-tolerant approach, combining the hardware performance with the software flexibility. Furthermore, according to [HEN13], "*Runtime adaption will be another key component to increase the reliability of future on-chip systems*". If we employ only design-time techniques, designs would have to be more and more conservative as they would have to reserve more and more resources to provide some infrastructure to support failures of components. Consequently, applying only design-time techniques would render future on-chip systems infeasible from the cost point of view, since the failure rate will continue to increase with future technology nodes.

In this context, this Thesis proposes a layered approach for runtime fault recovery for NoC-based MPSoCs. The method is built on top of a small specialized network used to search fault-free paths, and an MPI-like software protocol, implemented at the transport layer, hidden from the application layer. The software protocol detects unresponsive processors and automatically fires

the path search network, which can quickly return a new path to the target processor. This approach has a distributed nature since a source processor uses only local information to monitor the path to the targets. Thus, there is no single point of failure in the system and the approach is scalable in terms of number of PEs. The proposed approach provides complete reachability, i.e. if there is only one fault-free path to the target, it will be found, even in the presence of multiple faults. At the application level, this Thesis presents a checkpoint and rollback mechanism enabling the recovering of an application if a PE fails.

1.1 Overview of the Thesis

This section exposes the need for a layered approach to achieve a FT MPSoC and summarizes each layer of the approach. For instance, let us assume that a fault occurs in the path between tasks A and B, mapped at different PEs, as illustrated in Figure 3. Since a conventional message passing library has no FT mechanism (e.g. timeout), a *Receive()* call can wait indefinitely if an unreliable communication layer loses the message. This blockage can increase the communication buffer's occupancy, blocking other pairs of communicating tasks, ultimately resulting in a system crash caused by a single fault. In addition, a faulty packet might be misrouted or occupy buffers indefinitely, and a faulty router might generate wrong signals to its neighbors (Byzantine fault) leading to invalid system state.

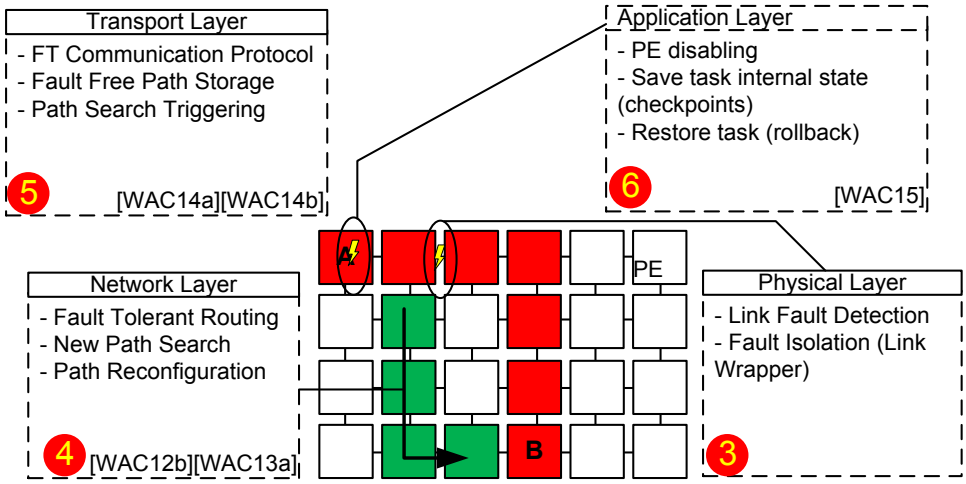


Figure 3 – Motivational example to justify the adoption of a layered approach for a fault-tolerant MPSoC. Red PEs represent the path taken by packets assuming an XY routing algorithm. Green PEs represents a possible new path to the destination. The numbers denote the Thesis' chapter detailing each layer.

Therefore, to avoid system crashes induced by hardware faults, the system must *detect*, *isolate*, and *avoid* the faulty region. For instance, if the fault is permanent and the network uses only deterministic routing algorithm, then the packet retransmission would use the same faulty path, generating a new undelivered packet. Thus, the system must have the ability to *locate the faulty router* and support *adaptive routing*, delivering packets using an alternative path and *avoiding* the faulty region. To solve this problem, we adopted a *divide and conquer* approach, where each layer is responsible for solving a given part of the problem.

The proposed layered approach is divided into four main layers. Each layer is numbered according to one Chapter of this Thesis. The first layer (Figure 3 – 3), named *physical layer*, is responsible for the fault detection mechanism and the wrapper module used for fault isolation purposes. The second layer (Figure 3 – 4), named the *network layer*, is responsible for replacing the original faulty path by an alternative healthy path. A fault-tolerant routing method that executes

a path search mechanism enables this process [WAC12b][WAC13a]. The third layer (Figure 3 – 5), named *transport layer*, implements a fault-tolerant communication protocol that triggers the path search in the network layer [WAC14a][WAC14b]. The last layer (Figure 3 – 6), *application layer* is responsible for moving tasks from the defective PE to a healthy PE, saving the task's internal state and restoring it in case of fault while executing a task (i.e. checkpoints and rollback) [WAC15].

1.2 Goals

The strategic goals of this Thesis include:

- Get a holistic understanding of FT NoC-based MPSoC design, from the physical level to the system level;
- Explore the use of routing algorithms for fault-tolerant NoCs;
- Explore the use of methods for isolation of faulty modules;
- Define a fault-tolerant communication protocol for NoC-based MPSoCs;
- Explore application recovery methods when faults are detected in PEs;
- Task remapping in the presence of faults in PEs.

To accomplish these strategic goals, the following specific objectives should be fulfilled:

- Propose a new fault-tolerant routing method for NoCs;
- Propose a method to isolate faulty modules;
- Reconfigure the network using a new faulty-free path;
- Change the communication protocol, adding fault-tolerant features on it;
- Addition of a method to detect undelivered messages;
- Propose a method for task remapping in case of faults in a processor executing a task.

1.3 Originality of this Thesis

The originality of this Thesis relies in the layered approach for achieving fault tolerance in MPSoCs. Each layer is responsible for solving a part of the problem, adopting a *divide and conquer* approach. At the network layer, we propose a novel fault-tolerant routing method, being the first in the state-of-the-art with full reachability and network topology agnostic. At the network layer, we propose and evaluate three approaches for delivering messages even in the presence of faults in the NoC. At the application layer, we present a context saving approach to isolate and restart faulty processors for a state of the art MPSoC.

1.4 Hypothesis to be demonstrated with this Thesis

This Thesis shows that a layered approach can cope with faults in the communication infrastructure and in the processing elements in such a way to ensure a correct operation of the MPSoC, even in the presence of multiple faults. The proposed methods increase the lifetime of current MPSoCs due to the isolation of defective routers, links, and processing elements. The remaining healthy components continue to execute the applications correctly.

1.5 Structure of this Document

This Thesis adopts self-contained Chapters, i.e., each Chapter presents a revision of the state of the art, the approach proposed by the Author and the results with the corresponding discussion. This approach is similar to the OSI standard, where each layer implements its functions and services assuming a lower layer. For example, the communication protocol (Chapter 5) assumes that there is a lower layer (Chapter 4 – Network layer) with a path search method.

The remaining of this document is organized as follows. Chapter 2 presents basic concepts and presents the reference platform used to validate the proposed methods. Next, Chapter 3 presents the physical layer, where network faults are detected and isolated. Chapter 4 presents the network layer, with the proposal of an FT routing method for NoCs. Chapter 5 presents the transport layer, with proposals for an FT communication protocol. Chapter 6 presents the last FT layer, the application layer. Finally, Chapter 7 concludes the Thesis bringing a discussion of strong and weak points of the proposed layered approach and providing some insights for future works.

2 REFERENCE ARCHITECTURE AND BASIC CONCEPTS

This Chapter discusses three main topics related to this Thesis:

(i) The basic features of the reference architecture, such as: tasks communication, task mapping and task execution;

(ii) Basic concepts of fault tolerance, focusing on fault tolerance at the system architecture level;

(iii) General assumptions for the fault-tolerant methods proposed in this work.

2.1 Reference Architecture

The reference architecture is the HeMPS MPSoC [WOS07][CAR09], which is an array of PEs interconnected by a mesh NoC [CAR10] (Figure 4(a)).

Each PE contains an IP connected to a NoC router, as illustrated in Figure 4(b). The IP has the following modules: (*i*) a 32-bit processor (MIPS-like architecture); (*ii*) a dual-port local memory; (*iii*) a DMA module, enabling parallelism between computation and communication; (*iv*) a network interface. The *Manager Processor* (MP in the Figure) controls the system and the applications. The MP also accesses the task repository memory, an external memory that stores the object codes of the applications. Slave processors (*SP*) execute user tasks. The hardware of the SPs and MP is the same, being the differentiation made by the software running at each PE. Each SP may execute multiple tasks and communicate with neighbors PEs through the NoC.

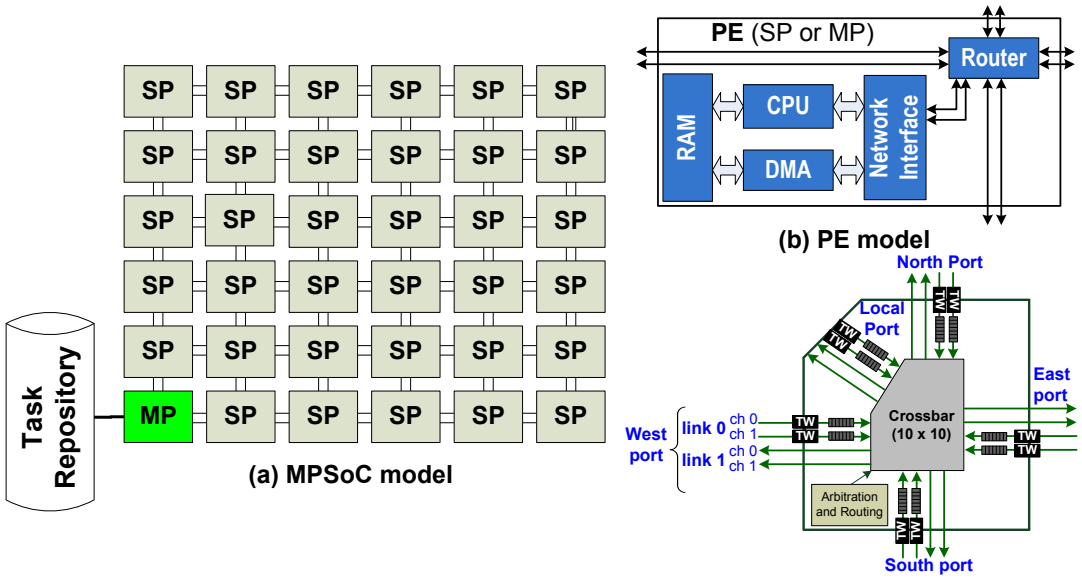


Figure 4 – MPSoC, PE, and Router models. The Manager Processor is labeled as MP and the Slave Processor as SP.

The Figure 4(c) presents the definitions for interconnections between routers used in this document. The wires (16 in our case) connecting one router to another are named *channel.* Each router contains two output channels and two input channels. The proposed router is composed of duplicated physical channels, being the two links combined named *links*. The aggregation of two links is named *port*.

The NoC adopts a 2D-mesh topology, with input buffering, credit-based flow control, and duplicated physical channels. The standard routing mode between PEs is the distributed XY

routing algorithm. The network also supports source routing such that it is possible to determine alternative paths to circumvent hot spots or faulty areas (feature discussed in details in Chapter 4). The NoC was originally designed for QoS purposes with duplicated physical channels and input buffers, as presented in Figure 5. As discussed in [CAR08], using duplicated physical channels compared to virtual channels (2 lanes) saves 12% in the area for a single router. For FT purposes, it is attractive to use duplicated physical channels. If a fault is detected at a given link (wires, input buffers or an output port), there is an alternative link capable to preserve the communication between routers.

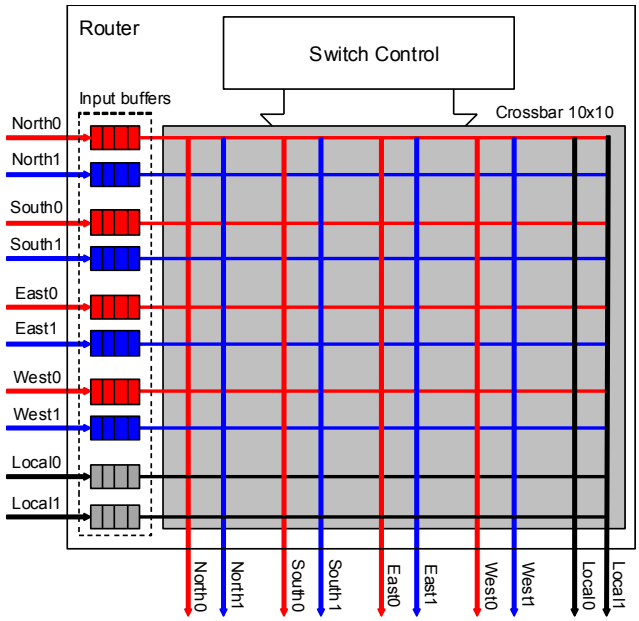


Figure 5 – Detailed view of the NoC with duplicated physical channel and input buffers [CAR09].

At the software level, each processor executes a small operating system (*kernel*), responsible for communication among tasks, management services, and multi-task execution. Message passing is the communication method adopted since it is more suited for distributed systems with local memory. Applications are modeled as task graphs A=<T,C> (example in Figure 6), where T = {t_1, t_2, ..., t_m} is the set of application tasks corresponding to the graph vertices, and C= {(t_i, t_j, w_{ij}) | (t_i, t_j) ∈ T and $w_{ij} \in \mathbb{N}^*$} denotes the communications between tasks, corresponding to the graph edges (w_{ij} corresponds to the communication volume between t_i, and t_j). The MPSoC assumes that there is an external MPSoC memory, named *task repository*, with all applications tasks (set T) loaded into the system at runtime.

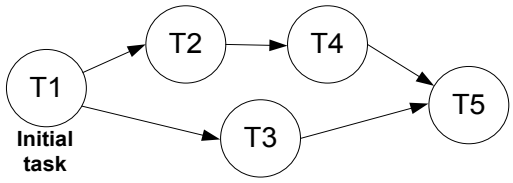


Figure 6 - Application modeled as a task graph.

2.1.1 Task Communication

Tasks communicate using two MPI-like primitives: a non-blocking *Send()* and blocking *Receive()*. The main advantage of this approach is that a message is only injected into the NoC if the receiver requested data, reducing network congestion. To implement a non-blocking *Send()*, a dedicated memory space in the kernel, named *pipe*, stores each message written by tasks. According to [TAN06], a pipe is a communication channel where messages are consumed in the same order that they are stored. Within this work, the pipe is a memory area of the kernel reserved

for message exchanging, where messages are stored in an ordered fashion and consumed according it. Each pipe slot contains information about the target/source processor, task identification and the order in which it is produced.

Figure 7 illustrates a communication between two tasks, A and B, mapped at different PEs. When task A executes a *Send()* (1), the message is stored in the pipe A, and computation continues (2). This operation characterizes a non-blocking writing. If the same processor contains both tasks (sending and requesting messages), the kernel executes a read directly in the pipe. In this example, task B mapped in other processor, executes a *Receive()* (3), and the kernel sends a message request through the NoC (4) to the sender. The scheduler puts task B in wait state (5), characterizing a blocking read. When task A receives the message request (4), the kernel set the pipe slot of the message as empty (6) and sends the message through the NoC. When the message arrives at task B (7), the kernel stores the message in the task B memory space, and task B returns to a ready state (8).

This communication protocol is susceptible to fail in the presence of faults. For example, if a fault is detected during the message delivery, task A will fill the communication path with flits, and will be blocked due the fault. Then task B will wait indefinitely for the remaining of the packet, ultimately blocking the application.

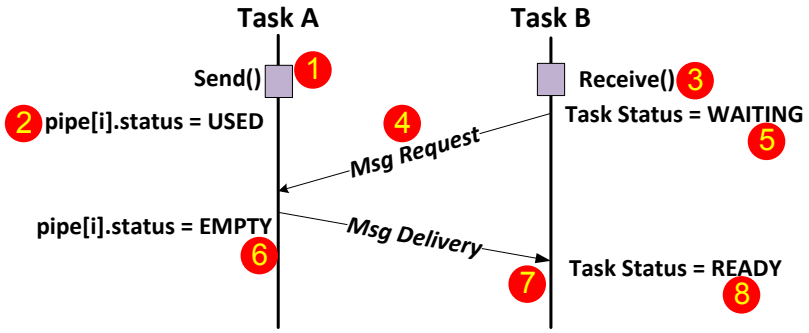


Figure 7 – Communication protocol adopted in the reference MPSoC.

Figure 8 expands the previous Figure to illustrate basic concepts used to measure latency in this text:

- Network latency, time spent on a single packet transfer, from the moment it is injected into the network hardware until it reaches its destination. This time is typically measured in tens to hundreds of clock cycles.
- Protocol latency, time to complete a packet transaction. This transaction may consist of multiple packet transfers, thus, the protocol latency consists of few network latencies, plus the time spent by the kernel to handle each packet transfer. This measure is typically few thousands of clock cycles.
- Application latency, time between two transactions, including the application computation time.

Each type of latency is subject to several sources of variability. For instance, network latency is subject to network congestion, protocol latency is subject to software interruption time and CPU load, and the application latency is subject to the amount of computation per packet transaction.

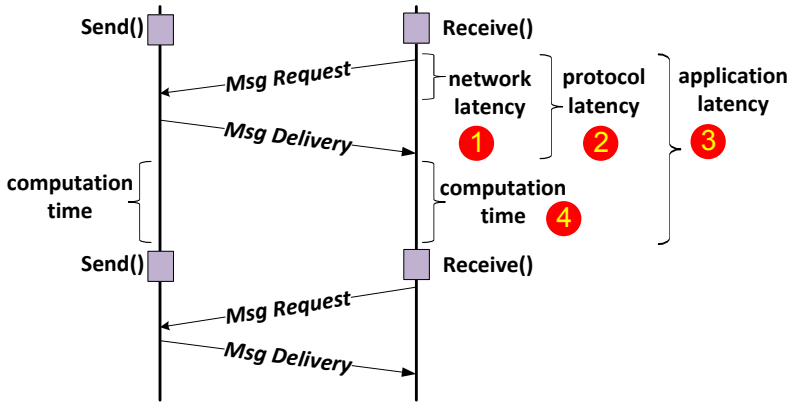


Figure 8 – Basic performance concepts in distributed systems that will be used in this Thesis.

2.2 Basic Fault Tolerance Concepts

To understand how errors can happen in MPSoCs, we first have to understand how they can happen in integrated circuits. The three terms: *defect*, *error*, and *fault* are related to system failure and thus need to be defined.

In [BUS05], a **defect** is defined as: "*A defect in an electronic system is the unintended difference between the implemented hardware and its intended design*". For example, there is a set of possible common defects in integrated circuits, such as:

- Process Defects – missing contact windows, parasitic transistors, oxide breakdown;
- Material Defects – bulk defects (cracks, crystal imperfections), surface impurities;
- Age Defects – dielectric breakdown, electromigration;
- Package Defects – contact degradation, seal leaks.

In today's technology, we can observe a growth in the defects due to aging and manufacturing process which are caused by the miniaturization of the transistor size [STA11]. Effects that were not relevant in past technologies are happening more frequently today, such as electromigration (EM) and negative bias temperature instability (NBTI), and even created new problems such as positive bias temperature instability (PBTI) [MAR13].

The defect can lead to an **error**, defined as: "*A wrong output signal produced by a defective system is called an error. An error is an "effect" whose cause is some "defect.*" [BUS05]. Note that a defect may or may not lead to an error. For example, a short to ground at an input of a AND gate do not show an error if the input is zero.

[BUS05] also defines **fault** as "*A representation of a "defect" at the abstracted function level is called a fault*." Note that the difference between a defect and a fault is rather subtle. They are the imperfections in the hardware and functions, respectively. Some examples of the failure modes resulting from these defects are opens, shorts, leakage, voltage threshold shift, variability in mobility.

The diminishing reliability of very deep submicron technologies is a fact. Moreover, it is widely accepted that nanoelectronic-based systems will rely on a significantly lower reliability rate than what was known so far. The probability of these defects is likely to increase with technology scaling, as a larger number of transistors are integrated within a single chip and the size of the chips increases, while device and wires sizes decrease.

Reliability and fault tolerance are very important requirements in current and future SoCs and MPSoCs. A *reliable* SoC provides detection and recovery schemes for manufacturing and

operational faults to increase not only system reliability and dependability, but also yield and system lifetime [COT12].

2.3 General assumptions related to the architecture and fault model

The present work adopts a set of assumptions related to the platform to receive the proposed fault-tolerant mechanisms. These assumptions can be divided in the MPSoC model and the NoC model. This Thesis adopts common features found in MPSoCs [JAV14][GAR13]:

- Each PE contains at least one processor with a private memory. This limitation is bind to the FT NoC approach presented in Chapter 4 where the fault-free path processing requires a programmable PE running the FT software protocol;
- Applications are modeled as task graphs and the communication model is message passing;
- A online mapping function maps tasks onto PEs, being possible to have more than one task per PE due to a multi-tasking operating (OS) system – this OS is referred as *kernel* along the text. The online mapping function is required to efficiently decide the new location on the MPSoC of tasks affected by faulty PEs;
- Fault-free *task repository* memory, i.e., the memory storing the object codes is assumed protected against errors;
- The Manager Processor is assumed to faulty-free.

Concerning the NoC features, we adopt the following common features found in NoCs [AGA09]:

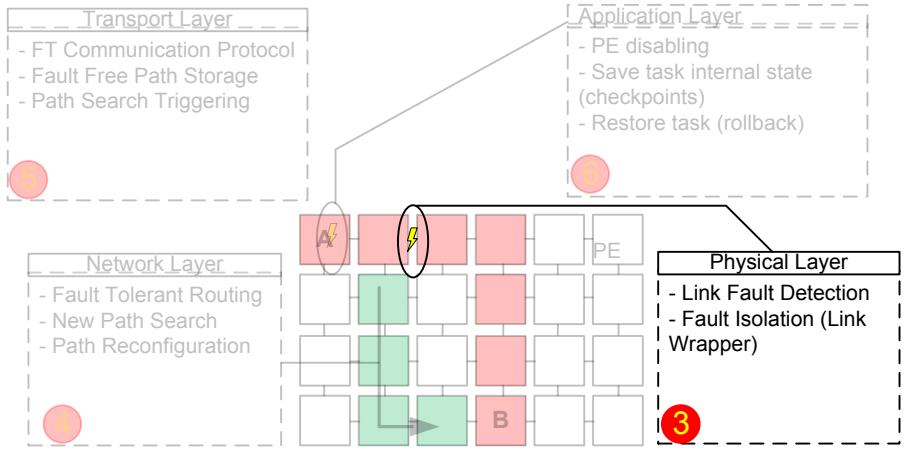
- 2D-mesh topology;
- Wormhole packet switching;
- Adaptive routing algorithm;
- Duplicated physical channels.

Therefore, platforms with similar architectural features may easily adopt the proposed fault-tolerant layered approach.

Testing is at the same time, one of the most costly step in a fault-tolerant method and highly dependent of reliability constraints and availability of an application. For example, chip testing requirements for automobile or avionics are typically higher compared to consumer electronics. Thus, the Thesis is focused on the recovery step, providing a choice of any desirable method for testing and fault detection according to application needs. For this reason, the actual test of PEs and the NoC are out of the scope of this work. There are recent proposals that could be used to test the PEs [GIZ11][PSA10] and the NoC [LUC09][FOC15]. This paper assumes the existence of a test mechanism to detect faults, responsible to isolate the faulty region (by using the test wrappers presented in Section 3.2) and firing the fault recovery methods.

The present work assumes **only permanent faults**, detected at runtime. Transient faults are out of the scope of the present work, and the literature provides several methods to cope with them (as [LUC09][COT12][WAN06][PSA10][KER14]. Fochi [FOC15] extended the current work to cope with transient faults, enabling the network to operate in degraded mode.

3 PHYSICAL LAYER



This Chapter details the lowest level of the fault-tolerant method, the physical layer. This layer is responsible for detecting network faults, signalize faults to the neighbor routers, and isolate the faulty region.

This layer may contain the test mechanism. In this Thesis, the test is out of the scope, then we present only an overview of some proposals at the physical layer. The objective is to show that the architecture is ready to cope with a fault detection mechanism. Furthermore, we introduce the concept of the Test Wrapper module, which will be used in the upper layers.

3.1 Examples of Physical Layer Approaches

Fochi et al. [FOC15] presents a NoC test approach. Figure 9 presents the approach, which uses CRC decoders to detect faults. The router can receive CRC decoders in the following locations:

- Before the input buffer (CRC 1), with the objective to detect faults in the channel.
- After the buffer (CRC 2), with the objective of detecting faults in the buffer. The channel can be healthy, but a fault can change the state of a given bit stored in the buffer.
- At the output ports (CRC 3), for detecting internal errors of the router. When this kind of error is detected, it is considered as a critical failure. Moreover, in this case, the router should be disabled because the integrity of the packets cannot be guaranteed.

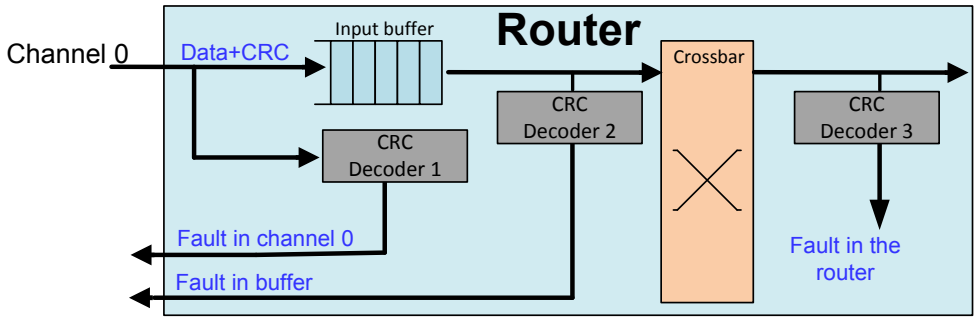


Figure 9– Fault-Tolerant Router architecture proposal.

It is very important to highlight that the CRC scheme does not guarantee 100% of fault coverage in the router. The CRC does not protect the control logic, and additional test mechanisms should be considered if higher coverage is required. For instance, [CON09] uses TMR in the state machines responsible for the buffer control.

The work in [FIC09] proposes fault tolerance at the physical level, with a mechanism for *port swapping*. Figure 10 on the left shows a router with a fault in the south input port and the east output port. In Figure 10 on the right, we can observe the configuration after the *port swapping*. The result is that three ports may be used, instead 2 before the port swapping. Matos et al. [MAT13] also presents a fault-tolerant scheme with not only the port swapping scheme, but also with fault isolation and a mechanism enabling to share buffers' slots.

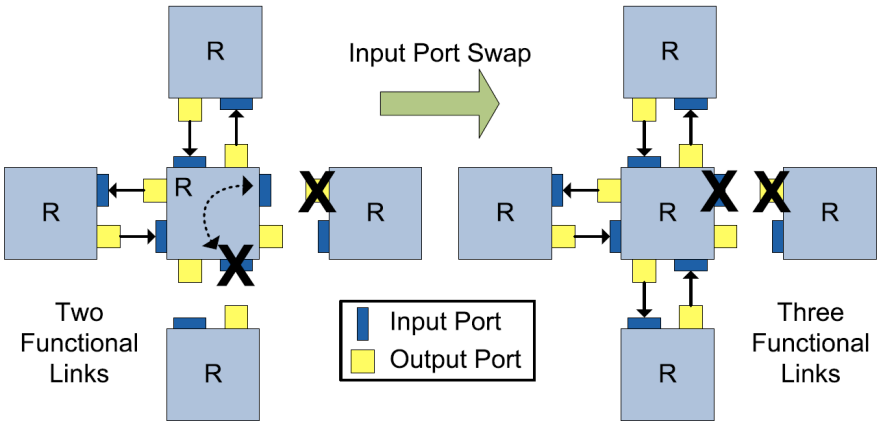


Figure 10 –Port swapping mechanism [FIC09].

Hébert et al. [HEB11] proposes a distributed fault-handling method for a NoC-based MPSoC. The Authors employ a heartbeat mechanism to detect faults in PEs. Each PE has a watchdog timer capable of detecting missing or corrupted heartbeat messages. Besides fault detection, the method also comprises faulty PE isolation and recovering via task remapping. Their main contribution is the *fault handler service* that checks the health status of a neighbor PE. To establish the health of an isolated core, the method compares the health register of the neighbor core with the last diagnosis result. Finally, if the PE is considered healthy, the last function is executed: un-isolate the healthy PE. This work focused mainly on the fault detection layer. Its main advantage is that transient and permanent faults are considered, since the core can be reused after a fault. In the present Thesis, once a fault is detected, the fault is considered as permanent. Such limitation is due to the scope of our work: implement fault tolerance from the network layer up to the application layer.

3.2 Physical Layer Proposal

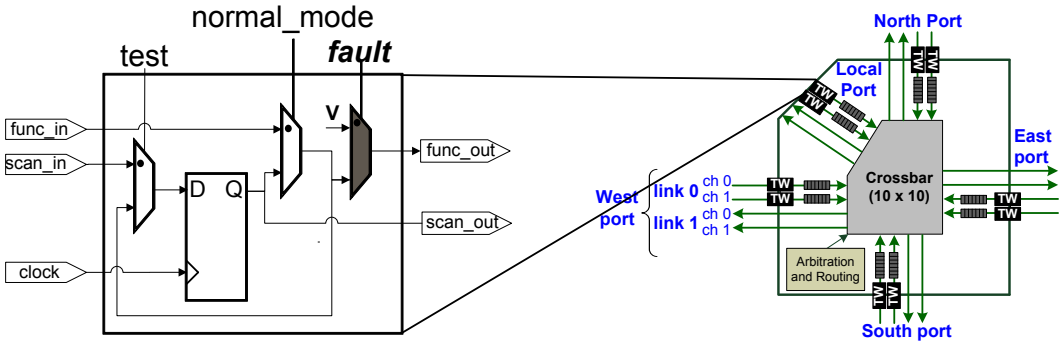
Complex SoCs, such as MPSoCs, need test wrappers such that each core can be tested individually to reduce both test data volume and test time [COT12][MAR09]. The basic element to build a wrapper around a given IP is the *test wrapper (TW) cell*. The main function of the TW cell is to switch between different operation modes. For example, in Figure 11, the three upper control signals (*test*, *normal_mode, fault*) signal controls the operation modes of the TW cell:

- Normal Mode - the wrapper logic is transparent to the core, connecting the core terminals to the functional interconnect, i.e., the input *func_in* is transferred to the output *func_out*;
- Test Mode - used for the actual test of the core. It configures the wrapper to access the core's primary I/O and internal scan chains via the test access mechanism.

In the scope of this Thesis, the main idea is to use existing test wrapper cells for the purpose of fault isolation. In order to support this, the TW cell includes an extra multiplexor, as illustrated in Figure 11(a) in dark gray, controlled by the *fault* signal. Activating the *fault* signal, the outgoing data of the TW cell (*func_out*) assumes the binary value '*V*' connected to the multiplexor.

With such approach, it is possible to isolate signals that are interconnecting two modules, avoiding faults propagations. The idea is that the test detection mechanism controls this *fault* signal, activating it if a fault is detected. The Figure 11(b) shows the locations of the TW in the router.

Note in Figure 11(b) that only input ports receive test wrappers. The reason is that the most complex module of the router is the input port, which comprises the input buffer and its control circuitry. The TW at the input port avoids the propagation of incoming data to other routers.



(a) Test Wrapper (TW) (b) TWs locations at the router

Figure 11 – In (a) the architecture of the test wrapper cell (TW), with an isolation mechanism controlled by the *fault* signal. In (b) the location of each TW at the router

The granularity of the isolation is an important design choice. The designer may:

- Isolate completely the PE, including the IP and the router;
- Control the isolation of the IP and the router separately;
- Control the isolation of the IP and the routers' links separately;
- Control the isolation of the IP and the routers' channels separately.

The main drawback of the first option is to disable the whole PE, even if it is healthy, when the fault is detected only the IP, but not the NoC. The second choice may lead to an unreachable IP, even if it is healthy, when the fault is detected at the router.

A trade-off that supports graceful performance degradation is to control the isolation of the faulty regions at the routers' ports, links or channels, individually. As the router has ten input channels, an individual fault signal may be used per channel. By abstracting the test method that detects the faulty modules, the designer choose the most appropriate test method, trading-off fault coverage, test time, and silicon area overhead.

In this Thesis, the isolation is tightly coupled with the FT NoC routing, which do not support a path search at the channel level. For this reason, when a fault is detected in the channel 0 of north link, for example, the whole north link is disabled (both channels). [FOC15] extended the current work disabling each link individually, making the channel to operate in "degraded mode", where packets are routed to the channel 1, if the channel 0 is faulty (and vice-versa).

Therefore, this Thesis adopts isolation of the links separately*. Even if one channel is healthy, the link is completely disabled.* Instead of having ten fault signals for disabling each channel, there are have five fault signals, one for each link.

Figure 12 illustrates the location of the TW cells between two routers. Note in Figure 12 that, for fault isolation purposes, TW cells are required only in the control signals (*tx* and *credit*). The *tx* signal signalizes incoming data. The output of the TW cell connected to *tx* signal, in the

presence of a fault in the input link, is ‘0’ (disabled). This action avoids the reception of new data by the faulty link (Figure 12 – 1), and consequently its propagation to other routers. The *credit* signal signalizes that the input buffer has space to receive new flits. The output of the TW cell, connected to *credit* signal, in the presence of a fault in the input link, is ‘1’ (Figure 12 – 2). This is a **very important** action since it enables the healthy router to drain the buffer connected to the output link transmitting flits to the faulty router. In this way, all flits transmitted to a faulty link are discarded, by keeping the credit signal enabled. The upper protocol layers will retransmit the discarded flits using a different healthy path, as described in the subsequent Chapters.

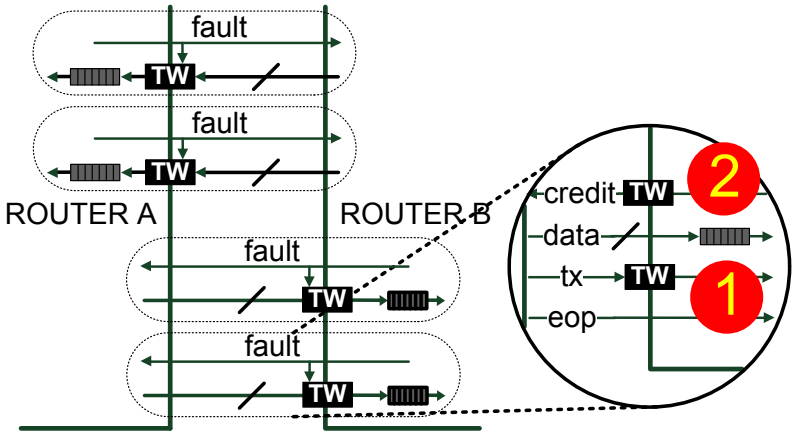


Figure 12 – Test Wrappers controlled by a fault signal.dfa

The IP connected to the two local links is responsible for writing in the local link fault signals. Therefore, by activating the *fault* mode of both local links corresponds to the isolation of the IP connected to the router, corresponding to a fault in the processor or another module of the processing element.

By activating a given fault signal of the East/West/North/South links, it is possible to degrade the router’s performance gracefully. This means that the router may still be used even if a given input link is faulty. If the router BIST detects faults in other modules, as the routing and arbitration logic, it may activates all five fault signals, completely isolating the PE to the rest of the MPSoC. Such complete isolation is necessary since the router is not able to transmit packets correctly.

TWs at the input links are sufficient to avoid fault propagation. Thus, it is not necessary to include TW at the output links. Figure 13 presents the location of the TW cells, as well as the connection between neighbor routers.

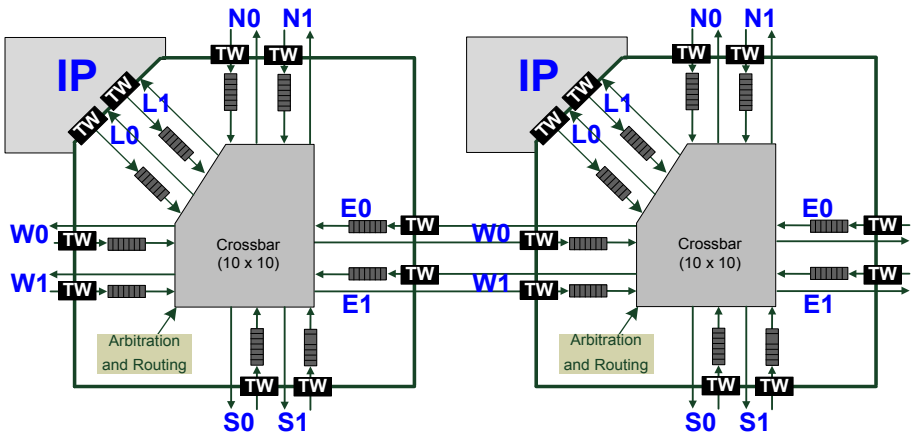


Figure 13 – PE architecture, detailing the internal router structure (TW: test wrappers).

3.3 Final Remarks

This Chapter presented the physical network level infrastructure that uses test wrappers to isolate faulty links. One important feature of the proposed method is the fact that the fault **isolation** is decoupled of the fault **detection**. In this way, the designer is free to choose any method available in the literature that is best suited to the design's constraints. Moreover, the designer is free to choose **where** the fault detection modules can be inserted.

This Chapter presented (according to Figure 14(a)) the option to cope with faults in three different locations:

- fault in the output port due to for example for some error in the crossbar of the left router;
- fault in the link itself (wires), e.g., crosstalk;
- fault in the input buffer, e.g. SEU (Single Event Upset) in some flip-flop of the input buffer.

The TW cell can isolate the link, avoiding the transmission of corrupted data. Figure 14(b) represents how the physical fault is represented graphically along this Thesis.

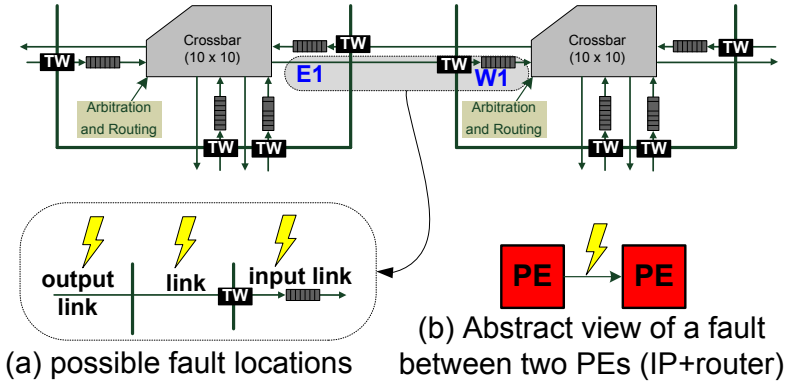
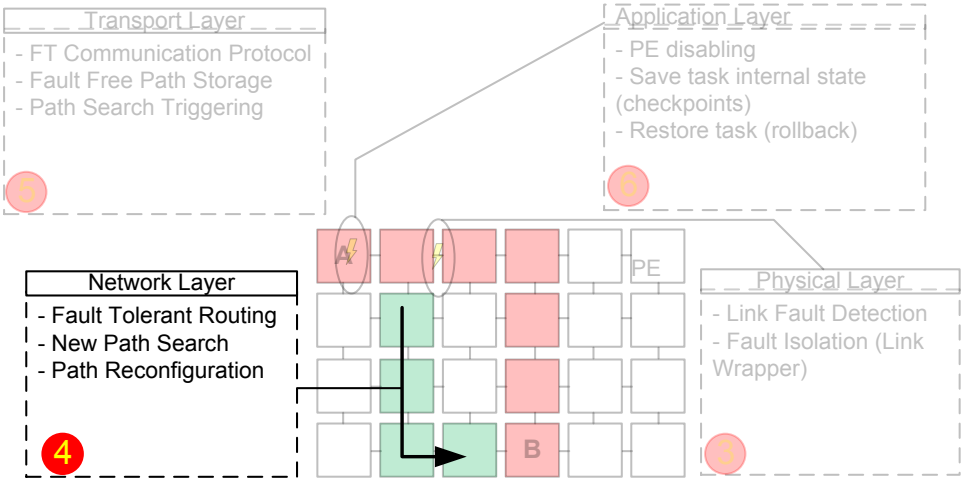


Figure 14 – Inter PE architecture, detailing the interconnections and fault locations in the router.

This isolation method requires minimal extra hardware and can be adapted to other router designs with different number of links. Moreover, it enables to reduce gracefully the network performance as the number of faults increases since it is possible to shut off one router link individually. The isolation method also ensures that the defective area does not disturb its vicinity, and any incoming packet is automatically dropped, avoiding fault propagation. Finally, the fault detection method (BIST circuitry) is *decoupled* from the rest of the fault recovery approach. The selected test method has just to write in the fault register when it detects a defective router link.

4 FAULT-TOLERANT ROUTING METHOD – NETWORK LAYER



This Chapter presents the state-of-the-art in the field of fault-tolerant NoC routing algorithms. Then, it presents the second layer of our FT MPSoC, which consists of the proposal of a set of requirements for FT routing algorithms and a novel FT routing method for NoCs.

4.1 State-of-the-Art in Fault-Tolerant Routing Algorithms for NoCs

Homogeneous NoC-based MPSoCs are a particular example of SoCs with natural hardware redundancy, with multiple identical processors interconnected by multiple identical routers. Therefore, if some processors or routers fail, tasks assigned to these processors may be migrated to other functional processors, and if a router fails, a new path in the NoC may be used. The Authors in [RAD13] survey the failure mechanisms, fault models, diagnosis techniques, and fault tolerance methods in NoCs.

Common approaches for fault-tolerant NoC can be divided in two categories:

(i) Add hardware redundancy to the NoC to deal with faults. Recently, Cota et al. [COT12] surveyed this category, which includes methods such as ECC and CRC codes, data retransmission, spare wires, TMR, backup path, and spare routers;

(ii) Add logic to enable the NoC using its natural path redundancy to find fault-free paths.

The first category is typically NoC-dependent, adding more hardware than the second category. This category can be categorized as static NoCs, with their parameters and structures fixed at design time [SHA14]. However, a general static NoC-based MPSoC cannot be efficient due to the dynamic behavior of applications running in MPSoCs, and the high probability of occurring faults in the chip during its lifetime. Our approach is classified in the second category, resulting in “dynamic” NoCs, due to its ability to reconfigure itself, avoiding faulty paths.

4.1.1 FDWR [SCH07]

Schonwald et al. [SCH07] propose an algorithm using routing tables at each router. The Force-Directed Wormhole Routing (FDWR) stores the number of hops to each destination for each port. This information enables to compute not only the shortest, but also all possible paths to all destinations. The algorithm updates the routing tables each time the network injects a packet. Before sending any packet, the router asks to all its neighbors the number of hops to the destination. Then, it chooses the neighbor with the minimum hop number and updates its table. Each router executes this process. Thus, the average time to send packets varies between 50 and

100 cycles, which is too long compared to non-fault tolerant NoCs (2 to 5 clock cycles). The Authors present results for 2D mesh and torus topologies. Due to the use of tables, any topology may be supported. The drawback of the approach is scalability since the table size increases with the NoC size.

4.1.2 Flich et al. [FLI07]

A similar approach to reduce the size of the tables, by dividing the NoC into virtual regions, was proposed by Flich et al. [FLI07]. The Authors propose a region-based routing mechanism that reduces the scalability issues related to the table-based solutions. The table-based routing requires an input port and the set of nodes associated with a given input – output port.

In a region-based approach, the idea is the combination of as many routers as possible in a group that shares the same routing path. Then, instead of having one table for each router, they can reduce the number of elements in the table and consequently reducing the area and performance overhead.

For example, in Figure 15, some routers have been grouped into regions (r1, r2, r3). The algorithm defines regions for the reference router (highlighted with a circle). All packets arriving in the reference router addressed to any destination included in region r1 necessarily need to use the W output port at the reference router.

From an initial topology and routing algorithm, the mechanism groups, at every router, destinations into different regions based on the output ports. By doing this, redundant routing information typically found in routing tables is minimized. Figure 15 shows a region grouping (r4 and r5).

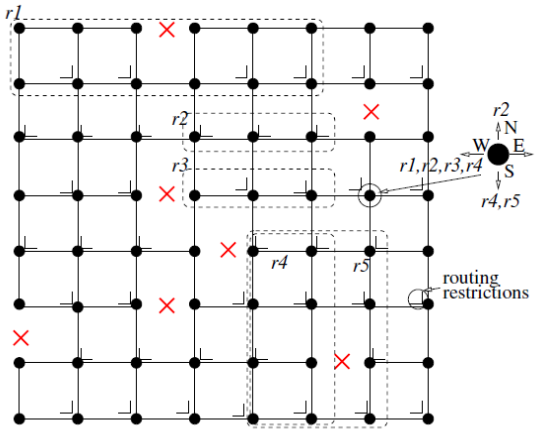


Figure 15 - Example of region definitions [FLI07].

The Authors divide the approach in hardware and software implementations. The first one is responsible for computing the faulty links of neighbor routers, and the route options using this information. After each router has computed its connections and possible routes, the software part can be applied to group different regions.

The Authors divided the algorithm for computing regions into sequential phases. In a first phase, the algorithm computes the possible set of routing paths for every source-destination. In the second phase, the algorithm computes the routing regions from the routing options. At every router the algorithm groups reachable destinations through the same output ports on the router and coming from the same set of input ports.

The Authors present throughput results in different faulty scenarios for different NoC sizes with a 2D mesh topology. As a conclusion, the region-based approach reduces the overhead induced by table's size but do not eliminate it.

4.1.3 FTDR [FEN10]

Feng et al. [FEN10] present a fault-tolerant routing algorithm, named FTDR, based on the Q-routing approach [BOY93], taking into account the hop count to route packets in the network. This table-based routing algorithm was modified to divide the network into regions, resulting in the FTDR-H algorithm. Compared to FTDR router, FTDR-H router reduces up to 27% the NoC area. When compared to a non-fault tolerant router, for a 4x4 NoC, the FTDR-H area overhead reaches 100%. Moreover, the number of tables increases with the NoC size. The Authors do not discuss the number of maximum faults that the algorithm supports, the supported topologies and reachability.

4.1.4 uLBDR [ROD11]

Rodrigo et al. [ROD11] present the uLBDR. The first version of the algorithm, LBDR [ROD09a], stores the connectivity status of the neighbors' routers (C_{east},C_{west}, C_{north} and C_{south}) and the turns ($R_{east\text{-}west}$, $R_{east\text{-}north}$, etc.) to reach its destination. It uses an adaptive routing for reaching the destination, using the connectivity status to use another path.

The problem with this approach is that it only ensures minimal path, leading to scenarios with unreachable routers. Then, if the neighbors' information is not sufficient to reach the destination, a *deroute* process (algorithm named LBDRdr) is executed, trying to route the packet in a non-minimal path. However, even with the *deroute* process full reachability is not ensured.

The Authors propose a third mechanism that ensures complete reachability of routers: the replication of the packets in two output ports: uLBDR. The drawback is that packet replication potentially increases the network congestion and a virtual cut through (VCT) approach is applied to avoid deadlocks. VCT requires larger buffers to store the entire packet and has longer network latency compared to wormhole.

The results show that compared to LBDR, the uLBDR increases 44.7% the silicon area. They also present results regarding power, frequency and flit latency. The presented results are for 2D irregular meshes, not exploring other topologies.

4.1.5 iFDOR [SEM11]

Sem-Jacobsen et al. [SEM11] propose two modifications in the original FDOR (Flexible Dimension Ordered Routing) algorithm [SKE09]: (*i*) create a boundary around a faulty router; (*ii*) reroute packets around the faulty router. Figure 16 shows a scenario where a faulty router is present (white router). The boundary around the faulty router is made reconfiguring its neighbor routers' as boundary routers. This way, any packet is routed around the faulty router.

Two virtual channels are used to ensure deadlock freedom by prohibiting one turn. The results have shown an increase of 279% in terms of area for a mesh NoC. The Authors do not discuss the reachability feature and the number of maximum faults that the algorithm may support. Moreover, a single fault disables the entire router, quickly reducing the network performance as the number of faults increases.

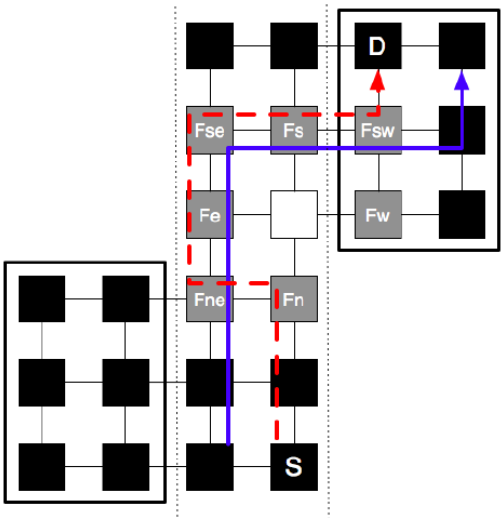


Figure 16 - The path followed by a packet around the fault (blue and red lines). The white node is the exclusion region, the gray nodes are the reconfigured as boundary routers.

4.1.6 Vicis [DEO12]

DeOrio et al. [DEO12] and Fick et al [FIC09] also propose fault-tolerant routing algorithms using tables. All routers contain a table to all router's destinations. The algorithm is divided in two steps. First, each router selects the direction to route a given destination based on its neighbors, updating the entry in its own table. Second, the router applies a set of rules to avoid deadlock. These rules disable turns without requiring virtual channels or duplicated physical links. On the other hand, this approach does not ensure full reachability. Presented results for mesh and torus topologies guarantee that at least 99,99% of paths are deadlock-free with up to 10% of faulty links.

4.1.7 Alhussien et al. [ALH13]

The Authors propose a fault-tolerant routing algorithm for NoCs. The algorithm is built based in a topology named DMesh [CHI11]. The topology is based in a mesh architecture, but the originality is a connection in the diagonal between routers as shown in Figure 17.

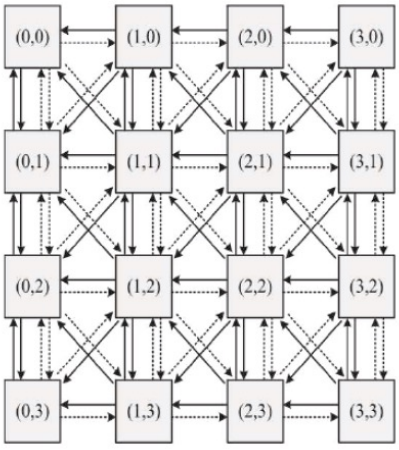


Figure 17 – DMesh topology.

The Authors developed a routing algorithm for supporting up to three faulty channels. The algorithm is based in the information of neighbor faulty links to avoid the faulty channels. They presented results for average latency for one, two and three faults for different traffic rates and patterns. Because the modifications are only made in the routing logic, the increase in router's area is only 7%. It is important to highlight that with more than three faults in the NoC the routing algorithm presents deadlock occurrences.

4.1.8 MiCoF [EBR13]

The method named MiCoF (Minimal-path Connection-retaining Fault-tolerant approach) is a combination of routing algorithm with a hardware support in the presence of faults. In the hardware scheme, the router connections are modified with the objective to maintain connectivity in the presence of faults. As shown in Figure 18 the east input channel is directly connected to the west output channel while the west input channel is connected to the east output channel. Similarly, the packets coming from the north or south input channels are directly connected to the south or north output channels, respectively. A similar approach was also taken by [KOI08].

The second contribution is a fault-tolerant routing algorithm to this architecture. This proposal takes into account the neighbor router fault information to route the packets. When a router is faulty, the links are simply connected to each other along the horizontal and orthogonal directions. Using hardware bypass (E→W, S→N, etc.) the Authors propose modifications in the routing algorithm. The route taken is modified according the fault location and destination. Figure 19 presents some scenarios with up to two faulty routers.

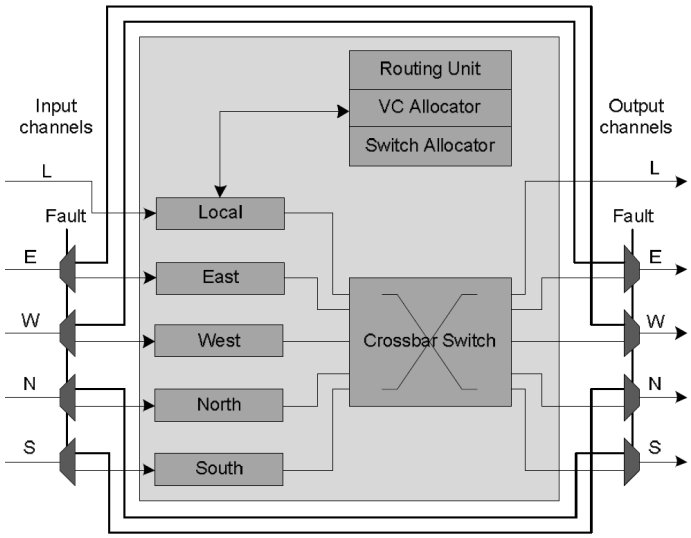


Figure 18 – Connection modification to maintain connectivity in presence of faults.

The algorithm showed a reachability of 99.998% in all combination of two faults in a 8x8 mesh topology, but maintaining the minimal path. Increasing the number of faults reduce the reachability percentage of the algorithm. The Authors showed that for six faults, MiCoF can guarantee 99.5% of reachability. The MiCoF area for a UMC 90nm technology is 7.295 mm².

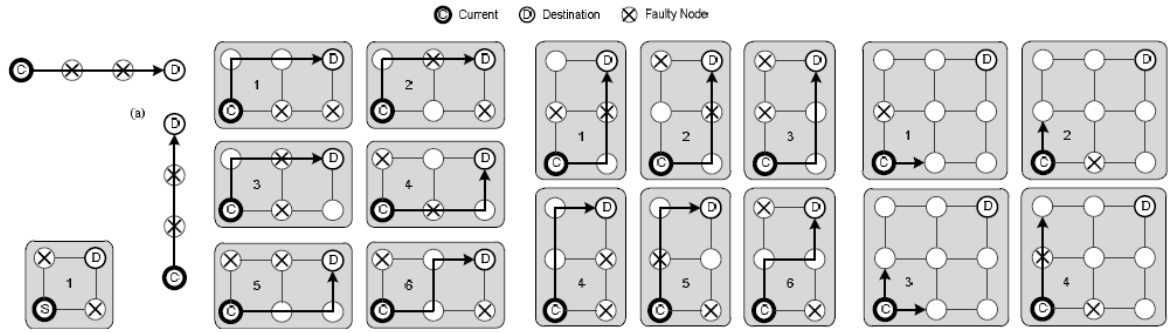


Figure 19 – MiCoF Routing in scenarios with two faulty routers [EBR13].

4.1.9 BLINC [DOO14]

The Authors propose modifications in a baseline router. Initially, routing tables are generated offline with all the routing is based on them. If a fault affects the network topology, BLINC utilizes a combination of online route computation procedures for immediate response, paired with an optimal offline solution for long term routing.

When the BLINC reconfiguration is triggered, two actions are started at parallel:

(i) a complete routing reconfiguration in software to generate new minimal routing tables for each router;

(ii) enabling pre-computed routing metadata, so to quickly resolve the affected routes. The rerouting response from BLINC is deadlock free, but not necessarily minimal.

To quickly find emergency routes upon a failure, while the complete routing reconfiguration is executed, BLINC employs compact and easy-to-manipulate routing metadata. The routing metadata consists in a region-based table (in the paper the term is segments). While the reconfiguration is executed, the routers within this region use this region-based table to route packets.

The evaluation shows more than 80% reduction in the number of routers affected by reconfiguration, and 98% reduction in reconfiguration latency, compared to state-of-the-art solutions. There is no discussion on the area overhead. The Authors only state that an 8x8 mesh requires at least 264 bits per router to store the metadata.

4.2 Discussion on the state-of-the-art

Evaluating the proposed works, we enumerate below features that should be present in a fault-tolerant routing algorithm, besides being deadlock and livelock free:

(i) **Generic:** being possible to adapt it to different NoCs, including different NoC topologies;

(ii) **Complete reachability:** i.e., if a path between a given source-target pair exists, the routing algorithm must find it;

(iii) **Granularity of the fault location and the moment it occurs:** i.e., faults may reach routers and/or links at any moment. Some routing algorithms restrict the number of simultaneous faults [DOO14] or if the fault occurs in links or routers [SEM11];

(iv) **Scalable:** the cost must be independent of the NoC size. Table-based approaches (discussed in the state-of-the-art) have their area cost linked to the NoC size;

(v) **Local path computation:** i.e., algorithms with a PE responsible to store and compute all paths must be avoided due to the large size of actual SoCs (a feature also related to scalability). Each router or PE must be able to compute the new path if a fault is detected, without interacting with a central manager, which can also be subject to faults;

(vi) **Acceptable cost:** i.e., the implementation of the proposal routing algorithm in a reference NoC should not severely impact the silicon area, power, and its performance.

We divide the FT NoC routing algorithms in two categories: (*i*) table-based approaches and (*ii*) distributed. In the first category, each router contains a table with two entries: target router and port. When receiving a packet, the routing module searches the port to forward it. This approach fits well with fault tolerance purposes because the table can be easily modified, changing the path to be taken to a given destination. The main advantage of table-based routing is that we can use any topology and any routing algorithm. The problem is that this approach is not scalable since routing tables are implemented with memories, also does not scale in terms of latency, power

consumption, and area. Thus, this approach is unfeasible for large NoCs [ROD11]. The number of destinations in this table is a function of the NoC size: a 3x3 NoC each router has nine destinations while in a 10x10 NoC there are 100 possible destinations. Some proposals for this approach can be found in the state-of-the-art [SCH07][DEO12][FIC09]. To solve this issue, some works propose to divide the NoC in virtual regions [FEN10][FLI07]. Then, instead of having a table with all possible destinations, some routers are grouped in "regions", reducing the number of destinations in the table. For example, Figure 20 shows the combination of routers in regions proposed by Flich et al. [FLI07]. Unfortunately, the main drawback of such mechanism is that, even with 16 regions, it still does not achieve full reachability and induces a long critical path [ROD09b].

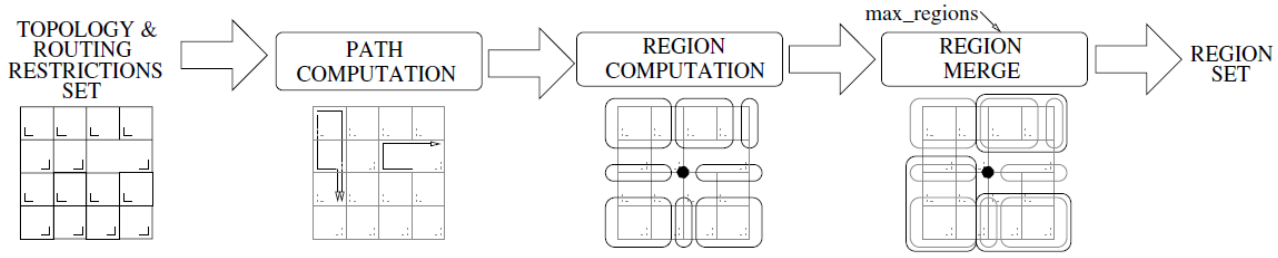


Figure 20 – Region computation process proposed by [FLI07].

Table 2 compares the reviewed approaches, positioning our method to the evaluated features. The closest approach is the uLBDR method, however the duplication of packets affects in the network traffic, increasing with the NoC size and needing a packet dropping method for the duplicated one. The fault-tolerant routing algorithms are a function of the NoC topology, as mesh and torus. Flich et al. [FLI12] evaluate topology-agnostic fault-tolerant routing algorithms, not specifically designed for NoCs, but the Authors claim that they may be used for NoCs. They present results regarding average routing distance, computing complexity, average packet latency, and throughput for mesh and torus topologies, with or without failures. However, results concerning how much time the algorithm takes to find the path are not presented. They do not present results for full reachability and silicon area too.

Table 2 – Comparison of different routing approaches.

Proposal	References	Routing Type	Fault Model	Area overhead	Scalable	Topology agnostic?	Full Reachability	max # faults	Offline Computation	Packet Dropping	VC/Physical Channel	Minimal Routing
uLBDR	**[ROD11]**	distributed	links/routers	46% (compared to the baseline router)	yes	irregular mesh	yes	any	yes	yes	no	yes
iFDOR	**[SEM11]**	distributed	router	~279% compared to FDOR	yes	irregular mesh	N/A	N/A	no	no	2 VCs	N/A
FDWR	**[SCH07]**	table	links/routers	N/A	no	yes	N/A	N/A	no	no	no	N/A
FTDR	**[FEN10]**	region based tables	links/routers	~100% in 12x12 NoC	no	N/A	N/A	N/A	no	no	no	N/A
Flich et al.	**[FLI07]**	region based tables	links/routers	240 gates in 8x8 NoC	no	N/A	yes (practically unfeasible)	N/A	yes	no	no	no
Vicis	**[FIC09] [DEO12]**	table	links/routers	300 gates per router (4x4), 330 (12x12)	no	irregular mesh /torus	no	N/A	yes	no	no	N/A
Alhussien et al.	**[ALH13]**	distributed	links/routers	7%	yes	no	no	3 links	no	no	2 physical vertical links	yes
MiCoF	**[EBR13]**	distributed	routers	7.295 mm² for a UMC 90nm technology	no	no	no	3 links	no	no	2 physical vertical links	N/A
BLINC	**[DOO14]**	table	links	N/A	no	N/A	yes	any	yes	no	N/A	yes
Proposed Work	**[WAC12b] [WAC13a]**	*path search*	links/routers	42% in LUTS/ 58% in FFs (wrt to the baseline router)	yes	yes	yes	any	no	no	2 physical	yes

4.3 Proposed Approach: MAZENOC Routing Method [WAC12b][WAC13a]

This section describes the **first original contribution** of this Thesis: a novel fault-tolerant NoC routing method. The method is inspired in VLSI routing algorithms [LEE61], and has the objective to find a path between source-target pairs in faulty networks. The method is generic, allowing abstracting the network topology. The method requires the following common features found in NoCs: wormhole packet switching; adaptive routing algorithm; duplicated physical channels. Features related to fault tolerance include:

(i) Deadlock avoidance mechanism;

(ii) Simultaneous support to both distributed and source routing;

(iii) Isolation wrapper cells at the NoC's input ports (Chapter 3);

(iv) Presence of a secondary path discover network (Section 4.3.1).

The use of duplicated physical channels ensures deadlock avoidance. The number of virtual or replicated channels required to avoid deadlocks is a function of the network topology. For example, two virtual or replicated channels are sufficient to avoid deadlocks in a 2D-mesh topology [LIN91]. Other topologies such as 2D-Torus and hypercube need additional channels [LIN91].

In the presence of faults, the path to circumvent faulty regions may require a turn that could lead to a deadlock. Therefore, a fully adaptive routing algorithm is required. Using the replicated channels, a 2D-mesh NoC may be divided into two disjoint networks, each one implementing a partial adaptive routing algorithm, for example, west-first and east-first [GE00], resulting in a fully adaptive routing. In fault-free scenarios, the distributed XY routing is adopted, and when faults are detected, source routing is applied using west-first and east-first partial adaptive routing algorithms to reach the destination.

This layer (Network layer) assumes that a higher-level layer, e.g. transport layer, triggers the proposed ***path discover procedure*** when a fault is detected. This can be done with the help of a fault-tolerant communication protocol or with hardware modules used to detect lost packets, as discussed in the next Chapter.

At the system startup, the network is assumed faulty-free, and packets are sent from the source PE to the target PE using the XY routing algorithm. The method, executed at runtime, comprises three steps, as illustrated in Figure 21: (*i*) seek new path; (*ii*) backtrack the new path; (*iii*) clear the seek structures and compute the new path. At the end of the described process, the source PE receives a valid healthy path to the target PE. This path is stored in the PE memory and the following packet transfers (from this source to the target) use this path. Thus, the proposed method is executed only once for each missed packet, without requiring the whole process for each packet transfer.

In the first step of the method, **seek step**, if the source PE identifies that it is not able to transfer packets to a target PE, this PE generates a seek request, asking for a fault-free path to the target PE. The router stores this request into an internal memory containing the fields S/T/P/#, meaning source router, target router, incoming port, and hop counter. Each router checks if it is the target router. If not, it increments the hop counter and broadcasts the seek request to its neighbors (except for the port that originated the request), via a separate network optimized for broadcast. The next routers store the seek request in their own internal memories, checking if they are the target router and repeating the process until the target router is reached or a timeout mechanism at the source PE is fired. In this case, the source PE broadcasts a clear command to free the seek memory and it tries another seek a random time later. Therefore, if a seek process is blocked due

to an overflow in the number of memory entries, it can be executed later. The target router is declared unreachable after a given number of unsuccessful seek requests.

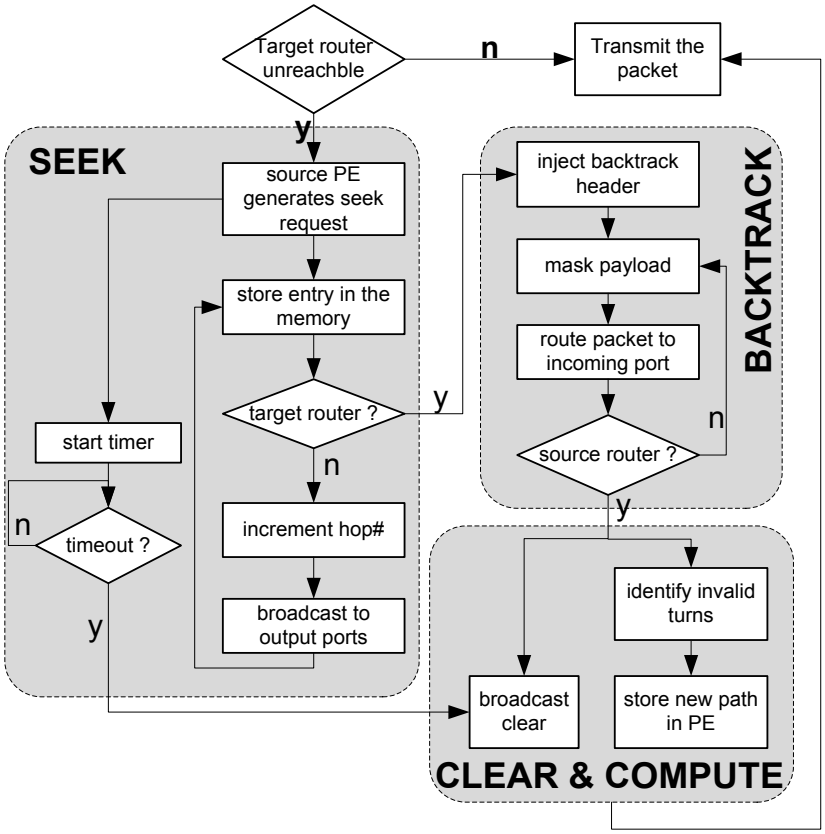


Figure 21 – Flowchart of the routing method.

The memory size is *constant* for the NoC design, regardless the NoC size, ensuring scalability. The number of the memory entries is only related to the maximum number of simultaneous seek requests, and its size is not proportional to the NoC size, as in the table-based routing methods. If the number of simultaneous seeks is larger than the memory size, the last seek waits the end of a seek request to restart the process. If the designer knows that it is unlikely to detect multiple faults at the exact same time, then it is possible to reduce the memory size to one. In other words, the designer can tradeoff between the time to determine a new fault-free path and the silicon area overhead.

Figure 22 illustrates a scenario with four faulty routers, one source, and one target router. Initially, the source PE requests a seek to its router, creating an entry in its local memory with value S/T/L/0, meaning that the source request came from the local port (L) and it is hop 0 (Figure 22(a)). Since this is not the target router, it broadcasts the seek request to the north and east ports. However, the east port points to a faulty router, which means that its input ports drain any incoming packet in order to avoid fault propagation, as explained in the previous Chapter. This way, only the seek request sent to the north port is propagated. The second router (Figure 22(b)) stores the entry S/T/S/1 in its local memory, meaning it received the request from the south port and it is the hop number 1. The broadcast continues until it reaches the target router (Figure 22(d)) via the east port with 8 hops.

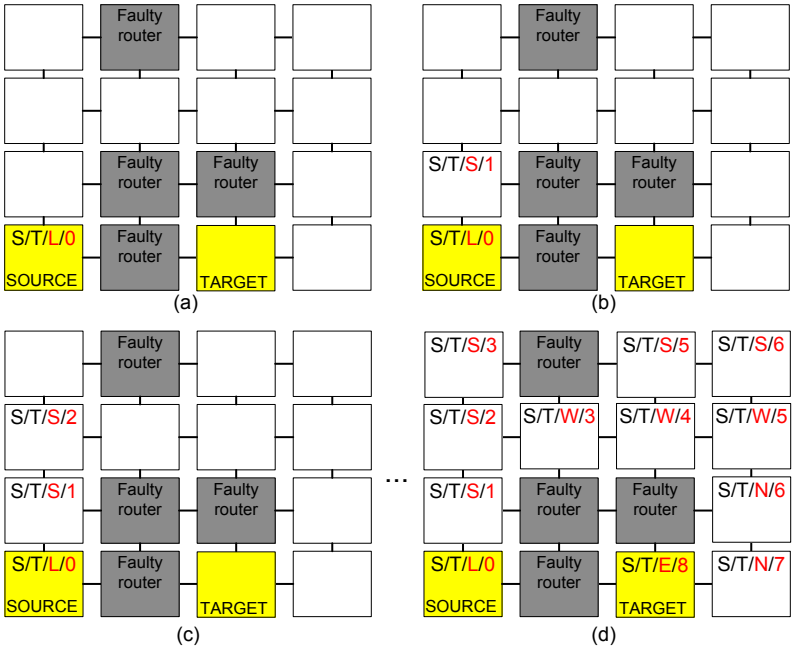


Figure 22 –Seek steps of the method in a 4x4 mesh, with four faulty routers.

The second step, ***backtrack step*** is executed only once, and starts when the target router is reached by the first received seek. All subsequent seeks are discarded. This feature ensures that, if there is only a single seek request at the moment in the network, the shortest path to the target is found. The target router injects a *backtrack packet* into the NoC, where its payload correspond to the ports selected by each router to reach the target router. The *backtrack packet* uses source routing, following the reverse path followed by the seek step. The backtrack ends when the source router is reached and the backtrack packet is sent to the source embedded processor to check the validity of the path.

Figure 23 shows the backtrack process, where the target router creates the backtrack packet. Its payload is initially filled with don't care data. At the first router (the target router), it inserts its seek incoming port (E) into the payload. Then, the packet follows the direction stored into the internal memory (E), reaching the next router. The next router also inserts its incoming port (W) into the payload. Then, the packet follows the direction stored into the internal memory (S), reaching the next router via its south port. In this way, the backtrack packet follows the reverse seek path, until it reaches the source router where the payload is now fully filled with the path to the target router. The path in this example is [N N E E E S S W E].

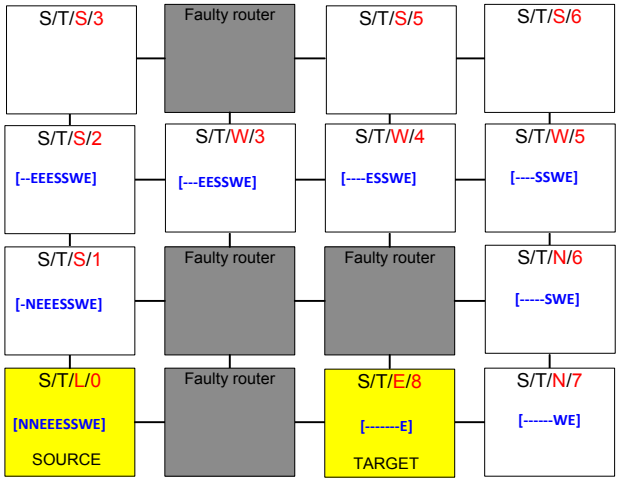


Figure 23– Backtracking process of the proposed method.

The backtrack packet may not be received in two cases: the target router is unreachable, or the number of maximal simultaneous seeks is exceeded. In the second case, the source router can initiate a new seek after a random time. If all the seek requests failed to receive backtrack, then the target router is declared unreachable.

The third step is named **compute path and clear seek**. The path followed by the backtrack packet might create a deadlock. For this reason, the source embedded processor checks the path, for instance [N N E EE S S W E], for invalid turns. When there is an invalid turn, it forces the packet to change the physical channel (or virtual channel), breaking the dependency cycle. For instance, the path presented above has a turn from south to west. This path should be changed from channel 0 to 1 because it is a prohibited turn in the west-first routing algorithm, but allowed by east-first one. Thus, the corrected path to the example target router must be [N0 N0 W0 W0 W0 S0 S0 W1 E1]. Once this path is computed, it is stored in the PE main memory for future packet transmissions.

In parallel to the path computation, the source PE requests to clear the entries in the seek memory labeled with the source router address. This clear command is broadcasted to the entire seek network, similarly to the seek process.

Note that the path computation depends on the network topology and its routing algorithms. For this reason, this part is implemented in software such that it can be easily modified. The hardware structure, presented in next section, is not dependent on the network topology and the routing algorithms.

4.3.1 Hardware Structure

Broadcasting messages in the NoC would create a congested scenario, disturbing the NoC performance. For this reason, a *seek routing module* was created, separating the NoC from the "seek network". Figure 24 illustrates how the seek routing module is connected to its neighbor routers (in this example they are N, S, W, E) creating two separated networks. The conventional NoC links are represented by dashed lines (number 2 in Figure 24) and the seek links are the continuous lines (number 1), consisting of 26 wires in this example. It is worth to mention that the seek links and the seek module are used during the seek/clear steps to broadcast the seek/clear commands without disturbing the NoC traffic (Figure 21), while the backtrack step uses the NoC.

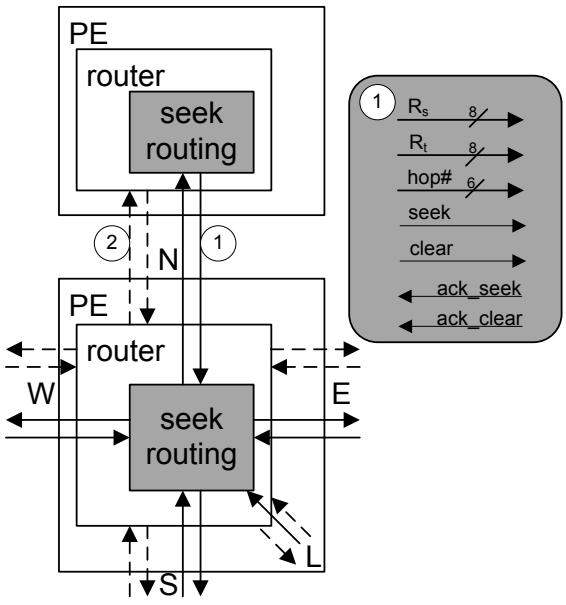


Figure 24 – Inter-router connection of the Seek router.

Note that the local port in Figure 24 has only the incoming seek link because, according to Figure 21, when a source PE detects an unreachable PE, it starts the seek process. The outgoing seek link is not required because the target PE does not receive data from the seek module.

Figure 25 details a generic router with 2 physical channels in the local input port. It also presents the logic required by the proposed method (gray area) and its intra-router signals. The inter-router links, presented in Figure 24, are not presented here for sake of simplicity. The routing and arbitration module is slightly modified to interface with the seek routing module. At the beginning of the backtrack step, the routing and arbitration module identifies a backtrack packet (1) and asks the seek module to search its internal memory (2) for the corresponding output port for the current seek request. The seek module answers with the appropriate output port (3). For the backtrack step, the payload receives the path information stored in the seek memory, through the added mux (4 and 5).

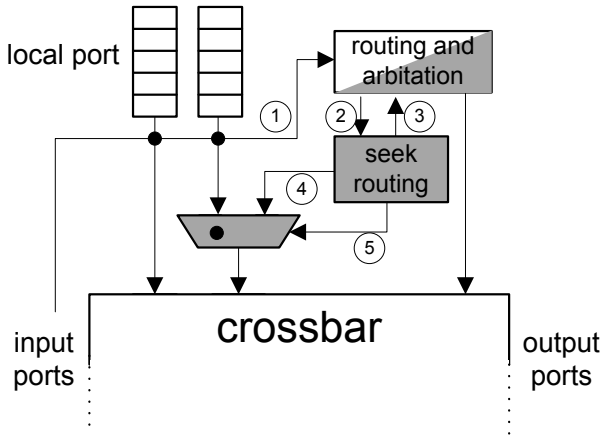


Figure 25 – Intra-router connection of the Seek router.

4.4 Results

In this section, the proposed routing method is evaluated in different NoC topologies. We also evaluate the Time to Find Alternative Paths (TFAP) and the area overhead induced by the modifications.

4.4.1 Experimental Setup

According to Salminen et al. [SAL08], from 66 NoC designs reviewed, the most common NoC topologies are mesh and torus (56%), bus and crossbar (14%), custom (12%), fat-tree (11%), and ring-based (7%). Since bus, crossbar, and fat-tree typically do not provide alternative paths in the presence of faults, these topologies were not evaluated.

The Spidergon STNoC topology is a regular [COP08], point-to-point topology similar to a simple bidirectional ring, except that each node has, in addition to links to its clockwise and counter-clockwise neighbor nodes, a direct bidirectional link to its diagonally opposite neighbor (Figure 26).

The Spidergon STNoC uses the Across-First routing algorithm. This algorithm moves packets along the ring, in the proper direction, to reach nodes which are closer to the source node, and use the across link only once at the beginning for destinations that are far away [COP08]. For example, a given packet from routers 0 to 5, would take the path 0→8→7→6→5. The Across-First routing requires two virtual channels to be deadlock free. The Hierarchical-Spidergon (Figure 27) topology uses two Spidergon networks layers, where each router communicates with its network and, additionally, with the lower/upper layer.

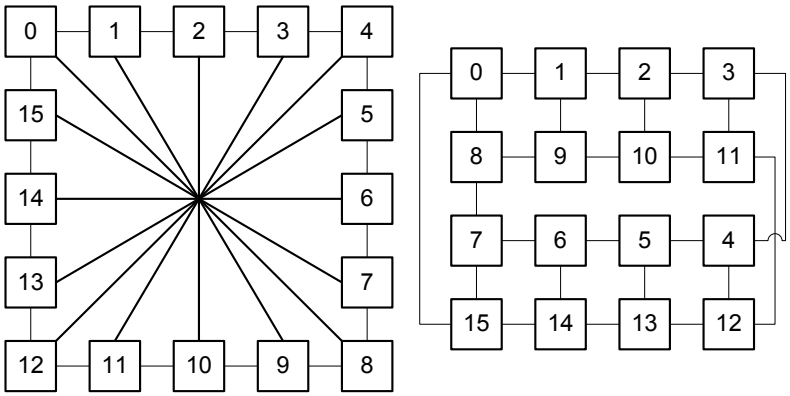


Figure 26 – Spidergon STNoC, topology and physical view.

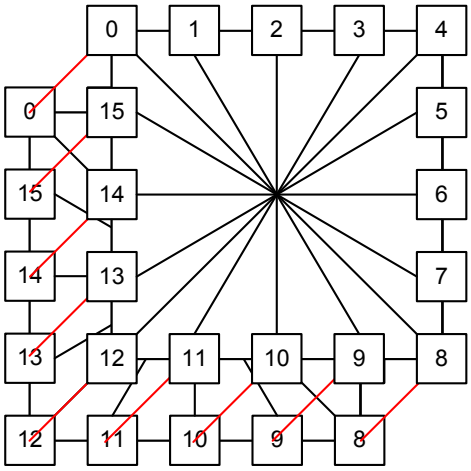


Figure 27 – Hierarchical-Spidergon topology. Red lines show the connection between the lower and upper layers.

This section evaluates the proposed routing method with four topologies: 2D-mesh, 2D-Torus, Spidergon and Hierarchical-Spidergon. All NoCs are configured with 100 routers, with four routers pairs starting the seek process. Figure 28 shows a configuration for the 2D-mesh and 2D-torus topologies.

90	91	92	93	94	95	96	97	98	99
80	81	82	83	84	85	86	87	88	89
70	71	72	73	74	75	76	77	78	79
60	61	62	63	64	65	66	67	68	69
50	51	52	53	54	55	56	57	58	59
40	41	42	43	44	45	46	47	48	49
30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Figure 28 – Configuration used in mesh and torus networks. Gray routers are faulty routers and the dark gray routers are communicating router pairs. Four communicating pairs are illustrated: 81→9, 45→47, 41→49, 9→55.

For instance, let us assume the communication from routers 81 to 9 using the torus topology. As illustrated in Figure 29, the proposed method is able to find shortest paths using the wraparound links 29 to 20 and 0 to 9, resulting in the path SEEEEEENEESSSSSSESSW (steps 1 and 2).

	90	91	92	93	94	95	96	97	98	99	
	80	81	82	83	84	85	86	87	88	89	
	70	71	72	73	74	75	76	77	78	79	
	60	61	62	63	64	65	66	67	68	69	
	50	51	52	53	54	55	56	57	58	59	
	40	41	42	43	44	45	46	47	48	49	
	30	31	32	33	34	35	36	37	38	39	
→	20	21	22	23	24	25	26	27	28	29	→
	10	11	12	13	14	15	16	17	18	19	
←	0	1	2	3	4	5	6	7	8	9	←

Figure 29 – Minimal path found by the proposed method from router 81 to 09 (dark gray), in a torus topology (gray routers are faulty node). The arrows indicate wraparound links.

The method was also able to find minimal paths in the Spidergon NoCs, assuming the same four communicating pairs used for mesh and torus. The Figure 30 illustrates the fault scenario for the Spidergon NoCs. Imagine that we have diagonal connections, e. g. router 0 is connected to 50, 1 to 51, etc. In the detail we have the four source – target pairs of evaluated flows.

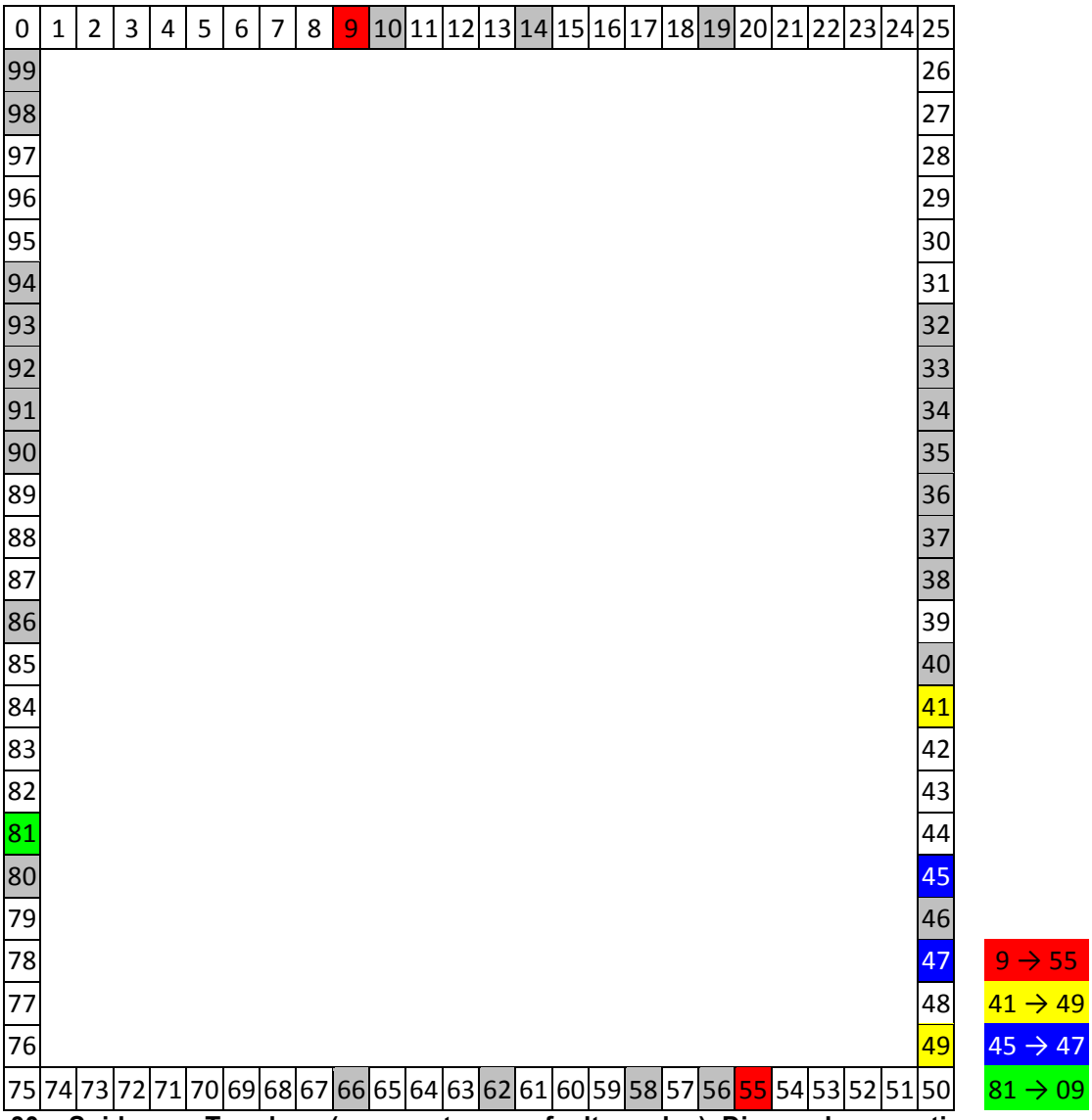


Figure 30 – Spidergon Topology (gray routers are faulty nodes). Diagonal connections were omitted. In the detail we have the four pairs of evaluated flows

4.4.2 Time to Find Alternative Paths (TFAP)

Table 3 shows the time to find paths in the presence of faults for each topology, in terms of clock cycles, for each step of the method. The values presented for step 3 include only the clean process, not the path computation. To account for the path computation time, we executed the path computation algorithm assuming a mesh topology and a MIPS processor running at the same frequency of the NoC. The result is that the processor takes, on average, 200 clock cycles per hop to evaluate the turns of the path. The complexity of this algorithm is O(n), where n is the number of hops in the path.

Next, we evaluate the average results from data presented in Table 3. The average number of clock cycles per hop (AHcc), assuming all topologies, is 16 and 15.5 for the seek step and the backtrack step, respectively. It is not required to account for the time of step 3 (clean) since it happens in parallel with the path computation. Therefore, the AHcc for step 1 and 2 is 31.3 (standard deviation of 2.6). Using these average values, we can estimate that, for instance, a path of 10 hops would take 2,313 clock cycles, being 313 clock cycles for steps 1 and 2, and 2,000 clock cycles to compute the path. To put this result into perspective, 2,313 clock cycles is a value inferior to a typical time slice in a multi-task operating system. Therefore, the impact to compute a new path represents a very small impact in the overall performance of an MPSoC system. Moreover, recall that, due to the locality of tasks in a MPSoC, two communicating tasks are expected to be close to each other. Thus, the number of hops is typically smaller than 10 hops.

Table 3 – Time to execute the proposed routing method, in clock cycles, for different topologies.

Topology	Rs→Rt	Number of hops	Step 1 *seek*	Step 2 *backtrack*	Step 3 *clean*
Mesh	09→55	43	688	548	635
	41→49	16	256	221	245
	45→47	16	256	221	620
	81→09	36	576	463	530
Torus	09→55	27	432	354	395
	41→49	16	256	221	245
	45→47	16	256	221	440
	81→09	20	320	269	350
Spidergon	09→55	5	80	89	500
	41→49	10	160	149	155
	45→47	4	64	77	65
	81→09	29	464	378	440
Hierarchical-Spidergon	09→55	5	80	89	246
	41→49	10	160	154	245
	45→47	4	64	81	230
	81→09	5	80	93	245

Once the path is determined by the proposed method, packets are transmitted as standard wormhole packets, with the same router latency compared to the base router [CAR10], i.e. the proposed method does not affect the network latency.

The reviewed works do not present similar analysis for path computation, so we can only perform a rough comparison. For instance, the Vicis NoC [DEO12] takes around 1,000 cycles to

reconfigure the network, without accounting the execution of the routing algorithm. In [SCH07] the average time between the sending of two consecutive packets is between 50 and 100 cycles, which increase latency. In our method this limitation does not exist, since the path search is executed once.

4.4.3 Multiple Seeks Evaluation

To evaluate the behavior of the approach with multiple seeks, we analyze a scenario with four routers simultaneously asking a new path. As an experimental scenario, we consider a 10x10 2-D Mesh, with a set of faulty routers, as presented in Figure 31. This scenario contains one bottleneck for the seek request at router 42.

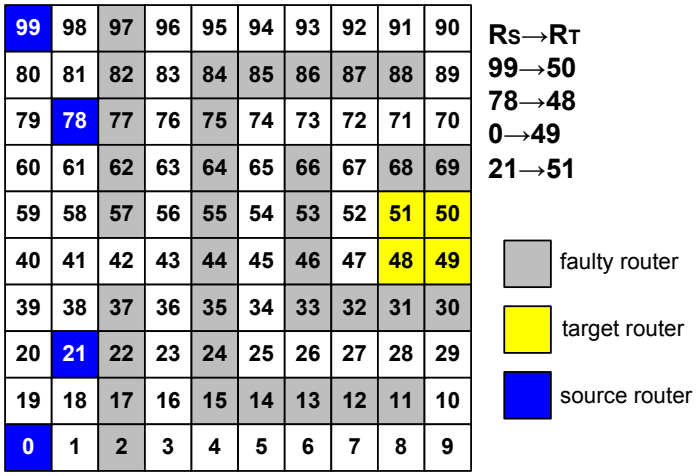


Figure 31 – Evaluation scenario of a NoC 10x10.

This algorithm returned the expected paths for each pair RS→RT, being all minimal. The bottleneck happens when the four requesting seeks arrive in the router 42. The seek module handles each request sequentially. However, each seek will be stored in a different entry in the seek memory, with the same output (east port). What happens is that each seek wave will find the minimum path, but the last processed seek will have a seek time (step 1) larger than the others.

In a scenario with five routers simultaneously asking a new path, the last seek request is not treated since there is no free entry in the memory. In this case, an upper layer should request another seek at a random time later, when there is space in the seek memory.

4.4.4 Area Overhead

Table 4 presents the area overhead in terms of look up tables (LUTs) and flip-flops (FFs) compared to the baseline router (with a seek memory with 4 entries). Regarding the LUTs occupation, the overhead is 42%, and for FFs 58%.

The router was also synthesized using Cadence Encounter RTL Compiler, targeting a 65 nm technology. The router with the seek module required 6,204 cells, occupying 43,049 μm². A silicon area comparison with previous works is not straightforward because the target router architecture might be considerably different and the technology library can be different. The best match is uLBDR [ROD11] router, which is similar to the evaluated router and uses the same technology. The uLBDR [ROD11] router consumed approximately 62,050 μm² and they do not present the features of the proposed method. Also, the presented baseline router is very small in terms of area. Even with the overhead induced by the seek module, the MAZENOC router (baseline router + seek module) is smaller than uLBDR.

Table 4 – Area overhead for Xilinx xc5vlx330tff1738-2 device.

	Area Occupation		
sub module	LUTs	FFs	module
Switch Control	351	97	baseline router
TOTAL	1610	433	
Seek module	370	184	baseline router modified + seek module
Switch Control	339	157	
TOTAL	2293	682	
Overhead	42%	58%	

Considering that the router represents less than 10% of the PE area, an increase of 50% in the router area represents less than 5% in the total area overhead of one PE, which is an acceptable cost considering full reachability even under severe faults scenarios, scalability in terms of NoC size, no impact on the communication performance after the path calculation, and topology independency.

4.5 Final Remarks

This Chapter presented an original method for fault-tolerant NoC routing, compliant with the following features:

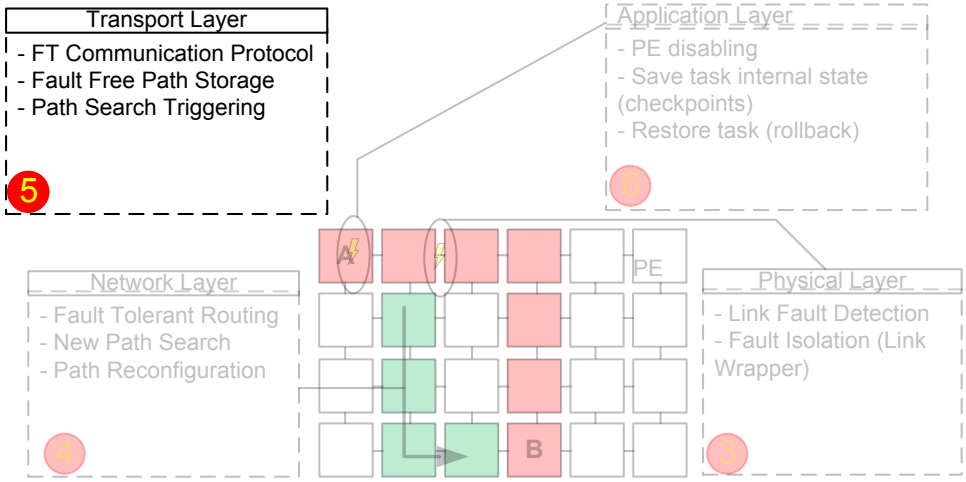
1. *Topology-agnostic*. the approach is divided into a hardware and software parts. The hardware part, i.e. the seek module, is topology agnostic. The deadlock avoidance algorithm is topology dependent, but it is implemented in software such that it can be easily modified. As far as we know, there is no other approach with this feature.
2. *Complete reachability*: most approaches support a limited number of faults and do not guarantee complete network reachability. The experiments presented in Section 4.4 demonstrate that the method is able to find a route even with dozen of faulty routers.
3. *Granularity*: the fault location can be any port link of any router.
4. *Scalability*: even though the proposed method uses a small memory, that has a completely different purpose compared to previous approaches (table-based), such that its size is not related to the network size.
5. *Fast path finding*: the method is able to find a new path in few hundreds of clock cycles per hop. This time is very fast compared to software execution time, context switching, interrupt handling, etc.
6. *No impact on the communication performance*: the path finding process uses the seek network for flooding, instead of the NoC. Once the new path is found, the packet latency behaves as a faulty free path assuming the same number of hops.

Moreover, the total path search and computation time is small compared to the literature, with an acceptable area cost. Due to these characteristics the proposed routing method may easily integrated into another NoC design that require fault tolerance.

Despite the advantages of the proposed method, the transmission of the backtrack packet uses the NoC to reach the source PE. Therefore, the path taken by the backtrack packet (from the target to the source PE) may execute invalid turns leading to a deadlock. This issue may be solved

adopting store and forward switching technique for these packets. This is possible because even for large backtrack packets the number of flits is small, being possible to store the entire packet in the routers' buffers.

5 FAULT-TOLERANT COMMUNICATION PROTOCOL – TRANSPORT LAYER



This Chapter presents the third layer of the proposed FT communication protocol. This layer is implemented in the kernel, hidden from the programmer. It triggers the path finding method presented in the previous Chapter when a message is not delivered. Section 5.1 presents and discusses the state-of-the-art in FT protocols for message passing. Section 5.2 and the following ones present original contributions of this Thesis.

5.1 The State-of-the-Art in Protocol and System Reconfiguration Approaches

Fault detection and location (i.e. fault diagnosis) are the first challenges in the quest for dependable NoC-based MPSoC. This step is not trivial since NoCs may have hundreds of routers. As surveyed in [COT12], the universe of online test approaches for NoC-based designed includes the use of error control coding schemes, BIST for NoC routers, and functional testing. These testing approaches have been extensively studied in the last decade and can be considered mature. Testing other parts of an MPSoC, such as processors and embedded memories can be considered even more mature.

The second challenge towards a dependable MPSoC would be the system reconfiguration. The MPSoC receives the fault location and changes the system configuration, masking regions with permanent faults. The predominant fault reconfiguration approaches are the use of spare logic (spare links, routers, or an entire PE), network topology reconfiguration, and use of adaptive routing algorithm [COT12].

As we move towards a macro architectural view of dependability in MPSoCs, two main communication paradigms must be taken into account: shared memory and message passing. Message passing is usually preferred in NoC-based MPSoCs due to the scalability offered by this communication protocol. Therefore, this Chapter adopts message passing as the inter-task communication method.

There are several attempts to implement message-passing libraries in MPSoCs, but most of them are focused on defining an efficient programming model for NoC-Based MPSoC in a layered communication protocol stack. For instance, Fu et al. [FU10] describe an MPI-based library where the task mapping is abstracted from the application. It also implements features such as broadcast-gather and message ordering. Mahr et al. [MAH08] argue that most message passing libraries for MPSoCs are specific for a single network. They developed a network-independent library that,

according to the target network topology, this library can be simplified or customized, removing the unnecessary parts and thus reducing the memory footprint.

Fault-tolerant message passing libraries were proposed in the context of fault-tolerant distributed systems, usually applied to a cluster of computers. An application may have very large runtime, and faults during this process may corrupt data or stop the application. Aulwes and Daniel [AUL04] include reliability features on MPI such as checksum, message retransmission, and automatic message re-routing. Batchu et al. [BAT04] test unresponsive processes by implementing self-checking threads that use heartbeat messages generated to monitor the MPI/FT progress. On top of this test approach, they built a recovery mechanism based on checkpoints.

The use of fault-tolerant message passing libraries specifically designed for MPSoCs targeting fault reconfiguration are scarce. For instance, Kariniemi and Nurmi [KAR09] present a similar fault-tolerant communication library targeting MPSoCs based on packet retransmission, where timers and CRCs are used to detect packet losses and bit errors caused by transient faults at the NoC links. In the case of a message lost, there is a retransmission mechanism triggered based on acknowledgment (ACK) and negative acknowledgment (NACK). The NACK message is generated after a timeout in the consumer, when the message is considered lost. Then the consumer sends a NACK message, requesting the retransmission. The Authors do not discuss if a faulty router is isolated or if it is circumvented in the case of a message considered lost. This may indicate a case when a packet could be always routed to a faulty router and will never be received. Furthermore, the Authors present the proposal, but without evaluating it.

Zhu and Qin [ZHU06] propose a fault-tolerant MPSoC executing a single program, multiple data (SPMD) environment. In the SPMD approach, all PEs executes copies of the same program. Then, if a tile is faulty, there is no instruction loss because another tile may execute the same program. The proposed framework uses an MPI-like message-passing library. The Authors evaluate a DSP application that has been replicated four times. This application was mapped in a 4x4 MPSoC, varying the number of non-faulty PEs, from 1 to 16. In the case of 16 available tiles, no faulty tile exists in the configuration. The Authors observe that the fault-induced performance degradation is smooth and predictable, with the exception of the configurations of very limited number of non-faulty tiles. The reason is that, with three or more functioning titles, one of the tiles is designated as the control tile while the computation tasks are mapped to the rest of the tiles.

Hébert et al. [HEB11] proposes a distributed fault-handling method for a NoC-based MPSoC. Faults at the PE are detected with heartbeat messages exchanged among PEs. Each PE has a watchdog timer capable of detecting missing or corrupted heartbeat messages. Besides, fault detection, the method also comprises faulty PE isolation and recovering via task remapping. As stated at the conclusion of [HEB11], the approach does not handle faults at the interconnect network. The results showed silicon overhead of 26% of the overall logic composing the MPSoC tile. The memory footprint overhead is about 12% when comparing to the baseline RTOS code.

Table 5 compares this review of prior work, showing that there are a few approaches addressing one or two steps toward the design a dependable MPSoCs (fault mitigation, detection, location, system reconfiguration, and system recovering). Approaches using a full MPI implementation target cluster of computers, excepting [MAH08], which does not present the implementation on a real platform. Most NoC-based systems adopt a partial MPI implementation due to memory restrictions. Two similar approaches to this work [ZHU06][KAR09] only take into account faults in the exchange of messages. As explained in the motivational example (Section 1.1), even retransmitting a message it will take the same faulty path, leading to an undelivered message. Besides the FT communication protocol, an important contribution of this Chapter lies in

coupling the layered fault-tolerant routing method with the communication layer. At the network layer, the proposed path recovery mechanism is capable of find fault-free paths with multiple NoC faults. At the communication layer, we trigger the path reconfiguration when an undelivered message is detected.

Table 5 – Comparison of different communication protocols adopting message passing.

Proposal	References	MPI Compatible?	Fault Model	Communication infrastructure
MMPI	[FU10]	YES simplified	NO	NoC
SoC-MPI	[MAH08]	YES	NO	NoC/Bus
LA-MPI	[AUL04]	YES	YES	Cluster of Computers
MPI/FT	[BAT04]	YES	YES	Cluster of Computers
MMP	[KAR09]	YES simplified	YES	NoC
Zhu et al.	[ZHU06]	YES simplified	YES	NoC
Proposed Work	[WAC14a] [WAC14b]	**YES simplified**	**YES**	**NoC**

Broadly speaking, the three first steps (fault mitigation, detection, location) seem to be mature, presenting different efficient test approaches for NoC-based systems. On the other hand, reconfiguration and recovery steps for MPSoCs are still under development. We believe that well-defined protocol reconfiguration/recovery approaches are a necessary first step before addressing the system-wide recovery (e.g. task migration and rollback-recovery) challenges. Thus, the contribution of this Chapter lies on: (*i*) triggering of the path reconfiguration step, where an alternative healthy path replaces the original faulty path; (*ii*) protocol recovery step, where the protocol retrieves data lost during a faulty data transfer. At the application layer, the application source code is not changed.

5.2 Proposed Fault-Tolerant Communication Protocol Description

When a given part of the NoC is faulty, eventually some packet will be not delivered. The **second contribution** of this Thesis is to modify the original communication protocol to detect these undelivered messages and to trigger the Seek Module. The required modifications to the existing communicating protocol were:

- All data packets are locally stored in the pipe before sending to the NoC. This feature enables the packet retransmission since the source PE temporally keeps a local copy of the packet;
- For all delivered packets, an acknowledgment packet is transmitted from the target to the source PE;
- Each packet generated by a single PE receives a unique sequence number.

Figure 32 details the proposed fault-tolerant communication protocol. Different from the original communication protocol presented in Figure 7 (page 25), the source PE does not release the *pipe* when a message is requested. The slot with the message being transmitted assumes the status "*waiting acknowledgment"* (label 1 in Figure 32). When the message is received, its sequence number is verified. If it is the expected sequence number, the task can be scheduled to run, the message is consumed (2), and the acknowledgment packet with the sequence number is

transmitted to the source PE (3). The last step of the protocol is to release the pipe slot, assigning to its position an *empty* state (4).

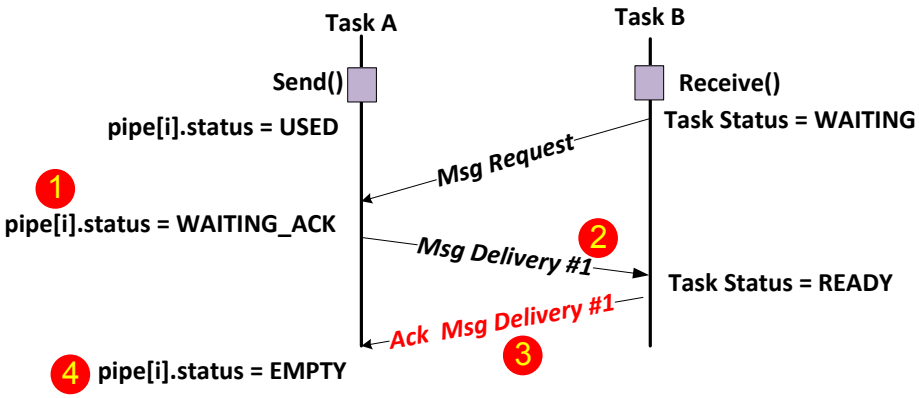


Figure 32 – The proposed communication protocol to provide FT.

5.3 Faults in the Proposed FT Communication Protocol

The proposed FT communication protocol can be interrupted in four main situations. This Section details each situation, explaining how the path search is triggered, and how the communication continues using the new path.

5.3.1 Fault in the Message Delivery

This scenario, illustrated in Figure 33, shows the sequence chart of the protocol when there is a fault in the path from A to B. Task B requests a message to Task A. Task A sends the message delivery packet and waits for the acknowledgment packet. An *Unresponsive Communication Detection* (**UCD**) scheme (described later in this Chapter) is used to detect that the acknowledgment is not received. Then the source PE declares the target router (Task B) is unreachable, and the fault-tolerant routing method is executed, and the faulty-free path stored in the source's local memory. After computing the new path, the packet is retransmitted to task B, and the acknowledgment is received by task A.

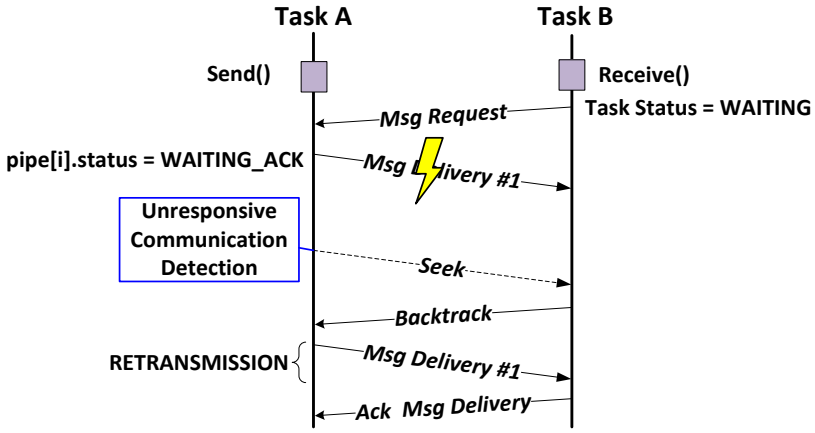


Figure 33 – Protocol diagram of a message delivery fault.

5.3.2 Fault in Message Acknowledgment

In this scenario, illustrated in Figure 34, task B requests message from A, task A sends the message to task B, and task B sends the acknowledgment back to task A. However, the acknowledgment is not received due to a fault in the path from task B to task A. In this scenario, task B successfully received the message, but the problem is that task A cannot release the *pipe* slot having the consumed message, since it assumes that the message was not received.

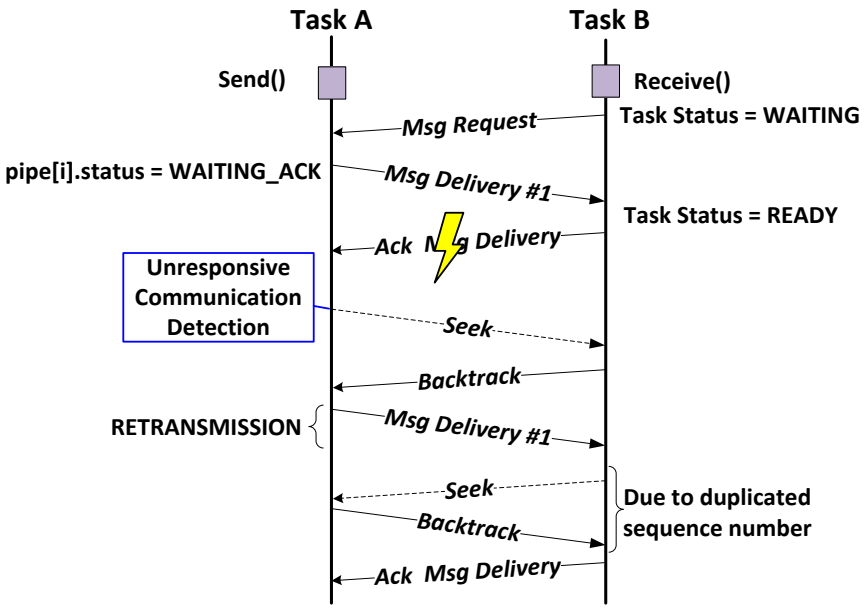


Figure 34 – Protocol diagram of ack message fault.

Therefore, from this fault, the UCD scheme interrupts the PE holding task A, inducing the process presented in the previous session (fault in the message delivery). As can be observed, the search for the new path would not be needed, since the path A→B is faulty-free, but it is impossible for task A to know why the acknowledgment was not received. This step corresponds to the "retransmission" label in Figure 34.

Task B, in this case, receives the same packet with the same sequence number. Task B discards this packet, signalizing an error in the acknowledgment path due a repeated sequence number. Then PE holding task B starts the seek process to the PE holding task A, to find a faulty-free path for the acknowledgment packet. Once the new path to task A is received, the acknowledgment packet is transmitted.

5.3.3 Fault in Message Request

Figure 35 shows the sequence chart of the protocol for a message request that was not received (scenario similar to Figure 33). After the fault detection (UCD scheme), the seek process is executed, and the last message request is retransmitted using the new path.

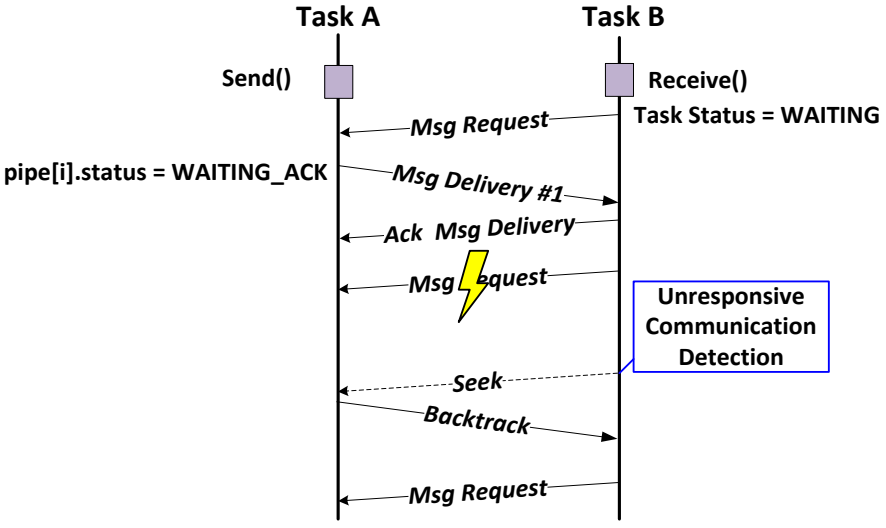


Figure 35 – Protocol diagram of message request fault.

5.3.4 Fault While Receiving Packets

In wormhole packet switching networks, a packet might use several routers ports simultaneously along the path from the source to the target router. If a faulty port is blocked while a packet is crossing it, the packet is cut in two pieces as illustrated in Figure 36. The result of the fault is two flows being transmitted inside the NoC: “F1” from task A to the faulty router; and “F2” from the faulty router to the task B.

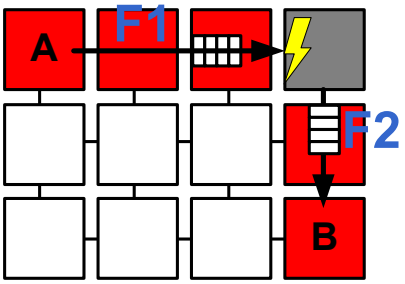


Figure 36 – Example of a faulty router in the middle of a message being delivered.

In fault mode, the wrapper logic around the faulty port signalizes that it can receive any flit transmitted to it. The practical outcome is that all flits of flow F1 are virtually consumed by the faulty port, avoiding fault propagation. This ensures that any data sent to a faulty port is immediately dropped, and F1, in this case, disappears.

The flits belonging to the flow F2 reach the target PE. The target PE starts reading the packet, and the kernel configures its DMA module to receive a message. The solution to discard the incomplete packet was implemented in the NI. The number of clock cycles between flits is computed. Here a fixed threshold was used, since the behavior of the NoC is predictable. If this threshold is reached during the reception of a given packet, the NI signalizes an incomplete packet reception to the kernel, and then it drops this packet.

5.4 Fault-Tolerant Communication Protocol Implementation

The proposed method targets *permanent* faults *on the NoC*. Assuming faults can block some ports of the router, all packets sent to these ports are lost. To avoid the loss of these packets, all packets supposed to use the faulty port must be rerouted. Therefore, we need to explore the redundant paths of the network with the proposed reconfiguration and routing method from the previous Chapter.

We also know that transient faults can lead to payload errors. These faults can be detected by online error detection codes, such as CRC [LUC09], and are out of the scope of our work. On the other hand, packets can be lost due to a misroute or by taking a faulty path. This chapter focuses on the second kind of fault effect.

At the application level the task code responsible for calling the execution of *Send*() and *Receive*() primitives does not require any modification. Therefore, the software designed for the reference platform may be used with the new fault-tolerant protocol without modifications.

The implementation of the FT communication protocol comprises two parts:

- ***Kernel***: transfers the messages from the task memory space to the kernel memory space and vice versa. This part implements the fault-tolerant communication protocol and the deadlock avoidance algorithm described in Section 4.3;

- ***Hardware***: The hardware modules related to the communication protocol are the network interfaces (NI) and the NoC routers.

The NoC received the path search module, presented in the previous Chapter. The idea is that when an unresponsive communication is detected, the path search method is triggered. The new path search is executed only once, when the unresponsive communication is detected. The search returns a new path to the source PE. This path must be stored for future packet transmissions, using source routing. In practice, each PE communicates with a limited amount of PEs, enabling to use small tables for path storage. Thus, the *path storage table* can be implemented either in hardware or software. If the path storage table is implemented in hardware, the best module to place it is the NI, since it simplifies the kernel. For instance, the kernel injects a packet to a given target in the NI. The NI checks the path table to verify if the target should be transmitted using XY (i.e. the path is faulty-free) or source routing (if the original path has faults) and, finally, the NI inserts the appropriate packet header.

The advantages to store the path in the NI include:

(i) the *network latency* for packets sent with XY or source routing is the same, reducing the performance penalties on faulty NoCs;

(ii) the number of entries in the path table is not a function of the NoC size, characterizing a *scalable* method;

(iii) the process to find a new path is fast. Once the new path is computed, the application's protocol latency returns to its original performance.

Next sections describe the three detection schemes to detect unresponsive communication developed during this Thesis. The **first** one, *Adaptive Watchdog Timers*, adopts a timeout to deliver the message. If a message does not arrive in a parameterizable amount of time, the message is set as lost, triggering the Seek Method. The **second** is named *Auto-detection of Faulty Paths*, and uses the *Seek Method* together with a small module to detect messages that are routed to faulty routers. After the message is detected, it sends a *Drop Detection* in the *Seek Network* to notify the producer that the message has been lost. The **third** implementation simplified the FT communication protocol, moving the acknowledgment to the next message request.

5.4.1 Adaptive Watchdog Timers [WAC14a]

The idea behind this approach is if any step of the communication protocol is stalled (e. g. a task do not receive a message) for a given number of clock cycles, this message is considered lost. Figure 37 represents six protocol transactions (in the *x*-axis) and their respective protocol latencies (*y*-axis), assuming faulty and faulty-free transactions. These protocol latencies are used to compute the AVG, the *average protocol latency*. K is a constant defined at design time per application. Once the threshold of *k*AVG* is reached, the proposed method to find a new path is fired and the packet is retransmitted using this new healthy path.

The AVG parameter is application dependent and computed at design time. The variability of the protocol latency defines the *k* parameter. Dataflow applications require a small *k* value because they a well-defined behavior. On the other hand, applications with tasks communicating with several other tasks present higher protocol latency variability, requiring higher *k* values. The user defines this parameter*, k*, based on an application profile.

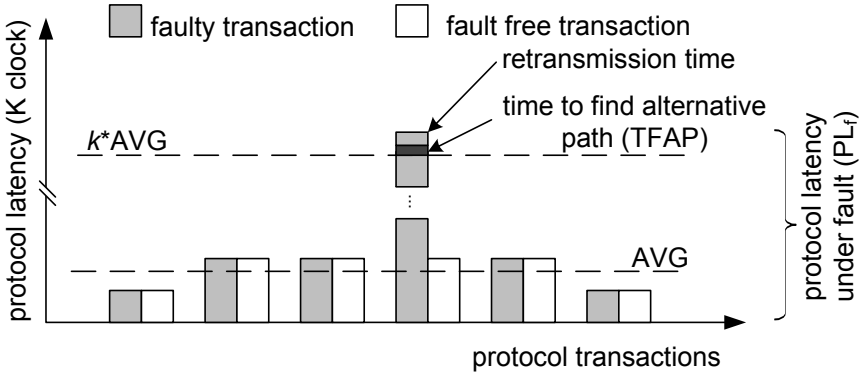


Figure 37 – Comparing faulty and fault-free protocol latency.

Figure 37 also illustrates the main components of the protocol latency when a fault is detected (PL_f). It can be seen that most time is spent in *k**AVG. Next, the proposed method is fired, spending a small amount of time to find the alternative path (TFAP). Finally, this new path is used for the packet retransmission. In addition, note in the last two protocol transactions that the protocol latency returns to the nominal value. The latency returns to the nominal values because the path calculation occurs only when the fault is detected, and the new path is used in all subsequent packet transmissions. Equation 1 summarizes the components of PL_f.

$$PL_f = k*AVG + TFAP + \text{retransmission time} \quad (1)$$

A fixed amount of time to start the seek process is not suitable for MPSoCs with applications inserted at runtime. A communication intensive application will stall for a long period if the fault timer threshold is too high. On the other side, a computation intensive application would generate false fault alerts since these applications can take a long period between communications. For these reasons, a new method is proposed, with an auto-detection of faulty paths, described in the next section.

5.4.2 Auto-detection of faulty paths [WAC14b]

The previous implementation may introduce a large overhead in the application execution time (AET). Section 5.5.1 shows that the use of watchdog timers may increase up to 68% the AET compared to a faulty-free scenario.

The goal of the second FT communication protocol approach is to reduce the AET overhead. The main idea relies on a hardware module responsible to notify the producer task that a given packet was not delivered. The producer then starts the *Seek Method* and follows the same steps as the previous approach.

The idea is to detect in the network when a packet is sent to a port that has been declared faulty by a given test method. When it happens, this test method has to block the defective port (or the entire router depending on the fault severity) by enabling the test wrapper presented in Chapter 3. When a neighbor router is about to send a packet to a faulty port, it fires a fault notification back to the source PE. Then, the source PE can start the Seek Method to find a new path. Figure 38(a) shows an example where a packet is sent from PE_1 to PE_2, but there is a faulty port in the middle of the path, next to the PE_3. Note that PE_3 is aware of its neighbor's faulty condition because of the 'fault' signals presented in Figure 38(b). PE_3 sends the packet to the faulty neighbor, knowing that this packet will be dropped and sends a fault notification back to the source PE (PE_1).

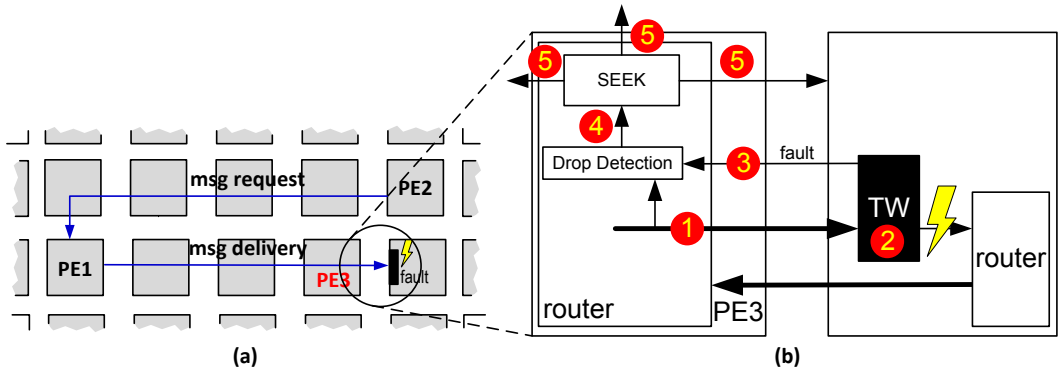


Figure 38 – Auto-detection of faulty paths process.

The highlighted routers in Figure 38(b) present the hardware module responsible for firing the *Drop Detection* message (the module is part of the router). PE_3 is sending a packet to the faulty router (label 1 in Figure 38 (b)). The Test Wrapper (label 2) discards this message and notifies the *Drop Detection* module of PE_3 that this connection is faulty (label 3). The *Drop Detection* module contains an FSM and registers to store the source-target addresses of the discarded packet. When a fault is detected, this module triggers the *packet dropped* process in the seek module (label 4). Then, the seek network broadcasts the Drop Detection message to the neighbors (label 5), addressed to the producer. When the producer receives a *Drop Detection* message, it sends a *seek clean* to free the tables in the seek modules, then it triggers the *new path discover procedure*.

Figure 39 details the FT communication protocol with path reconfiguration and message retransmission. The execution of *Send()* by a given task transfers the message to the OS level, into the *pipe* structure. The following sequence of events occurs:

1. The task requesting data (Task B in PE_2) injects a *message request* in the NoC.
2. The OS of PE_1 answers the request, through a message delivery packet. In this Figure, the message reaches a faulty router (fault in the west input port). The last faulty-free router, PE_3 (Figure 38 (b)), fires the fault recovery process.
3. The PE_3 router discards all received flits at the faulty port and uses the seek network to start a wave from PE_3 (*Drop Detection* message) to PE_1. This step is managed locally at the router, does not require any action by the processor. Thus, the wave reaches PE_1 in few clock cycles.
4. When the FAULT wave reaches PE_1, the processor is interrupted, and two actions are managed by the OS: (*i*) the ***Seek Network*** launches a CLEAN wave to free the slots of the seek memory used by the FAULT wave; (*ii*) the ***Seek Network*** launches a SEEK wave to reach PE_2.
5. The NoC is used to transfer the backtrack packet. At each hop, the packet receives the port identification from which the SEEK wave entered.
6. When the backtrack packet reaches PE_1, the ***Seek Network*** launches a CLEAN wave to free the slots of the seek memory used by the SEEK wave, and the OS computes a deadlock-free path. Using this example, the resulting path could be E0-E0-E0-N0-E0. This path is stored, and all subsequent communications between PE_1 and PE_2 use this new path based using source routing.
7. PE_2 receives the message, schedules task B to execute and sends the acknowledgment required by the protocol.

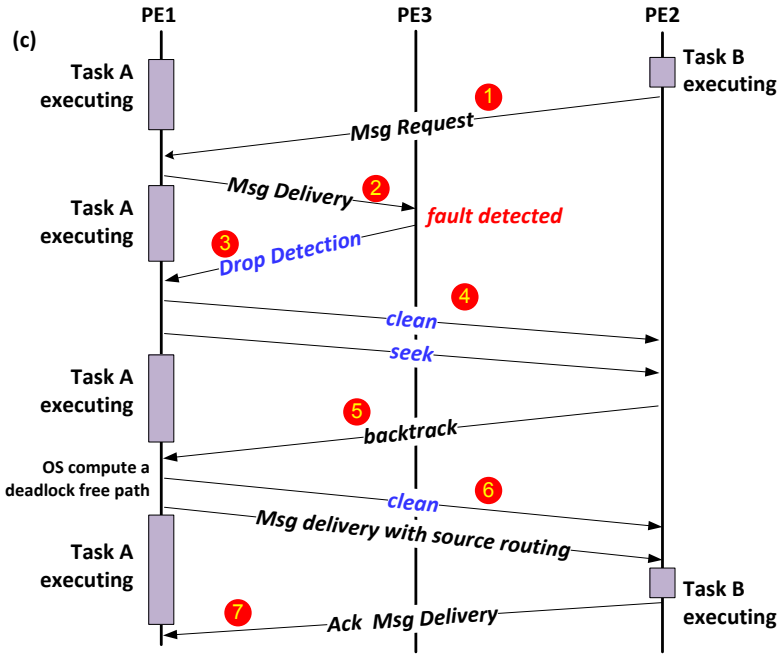


Figure 39 – Sequence chart of protocol communication with Auto-detection of faulty paths.

5.4.3 Auto-detection of faulty paths with simplified protocol

The third approach simplifies the previous protocol, by removing the acknowledgment step. *One can see that in a communication between two tasks, the message acknowledgment is always followed by a message request* related to the next packet. For example, in Figure 40(1) Task B acknowledges the reception of message number 1, then Task A frees the pipe in Figure 40(2). The next step (Figure 40(3)) is the reception of a message request, where the second message status is set as WAITING_ACK.

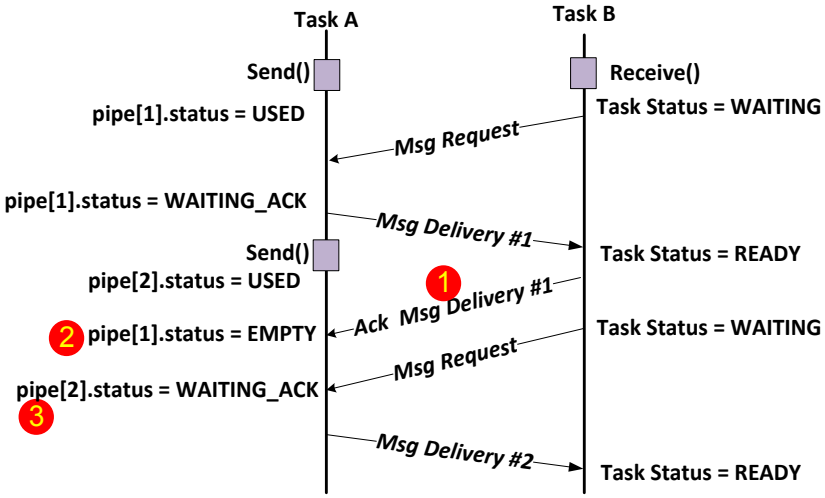


Figure 40 – Example of acknowledgment message followed by a message request.

This third implementation merges the acknowledgment message with message request, acknowledging the reception of a message and freeing the pipe slot. Figure 41 presents the new FT communication protocol. This sequence chart shows the message exchanging of two *Send() – Receive()* exchanges between Task A (t_A) and Task B (t_B).

1. t_A executes a *Send()*, storing in the position one of the pipe the first message.
2. t_B executes a *Receive()*, generating a message request to t_A.

3. t_A searches in all pipe positions for messages from t_A to t_B with WAITING_ACK status to change its status to EMPTY. There is no message with this status, because it is the first message to be sent.
4. Then, t_A searches for the requested message from t_A to t_B with status equal to USED to change its status to WAITING_ACK. This message (position one of the pipe) change the status from USED to WAITING_ACK and is injected into the network.
5. t_A executes a second *Send()*, storing the message in the second pipe slot.
6. t_B receives the message, and sometime later, executes a new *Receive()*, generating a second message request.
7. t_A receives the message request and again searches in all pipe positions for messages from t_A to t_B with WAITING_ACK status. One message fills this condition, and the pipe slot number 1 is released, going to EMPTY state. As the message request has been received, one can safely assume that the previous message has been received because the communication has not generated a *Drop Detection* message.
8. t_A sets the second message slot, with status USED, to WAITING_ACK, and inject into the NoC the second message to t_B.
9. The last step is to confirm the delivery of the last message of the task. The solution is to verify if the task has already finished. In this case, the pipe slot can be set to EMPTY.

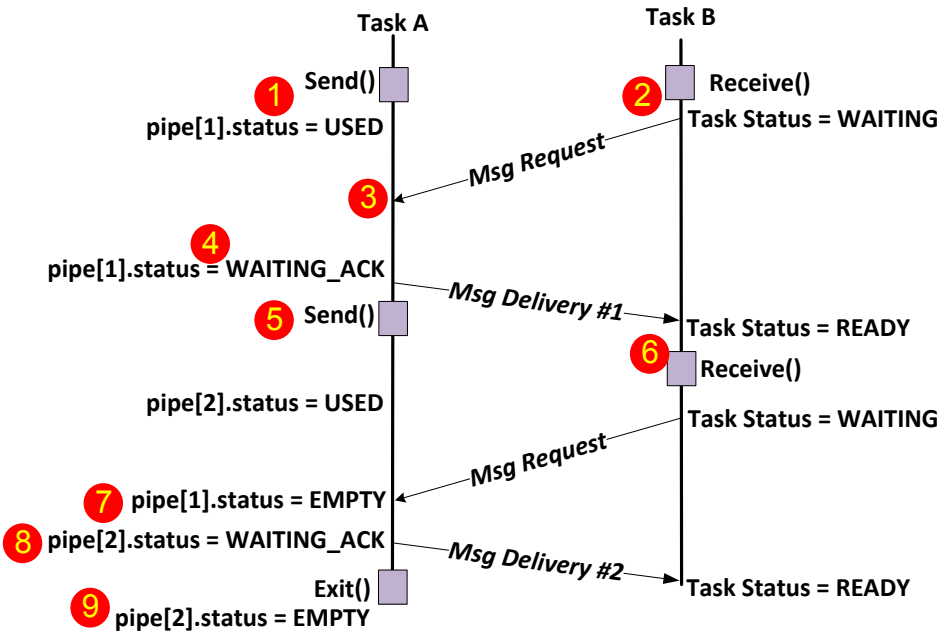


Figure 41 – FT communication protocol without the acknowledgment message.

5.5 Results

This section first presents the experimental setup, the evaluation flow, and the applications used as case study. Next, the results are divided into three sections, one for each method to detect lost messages. The last sections present the area overhead and compare the proposed methods.

By using the isolation method presented in Chapter 3, it is possible to disable any individual router link of the NoC. The router has five ports, but the local port is not used for fault injection, as explained in Chapter 3. For this reason, faults can be injected in the eight remaining router links. A small 3x3 network has 48 links (excluding the links in the chip's boundary) in total. The number of

possible fault scenarios grows exponentially with the number of simultaneous faults. For instance, 1 fault requires C(48,1) = 48 scenarios and 2 faults requires C(48,2) = 1128 scenarios. It is not possible to execute the evaluation manually since the number of fault simulations grows very fast. For this reason, an automatic and parallel fault analysis flow, illustrated in Figure 42, has been created [CAS14][PET12]. This flow is divided into five main steps.

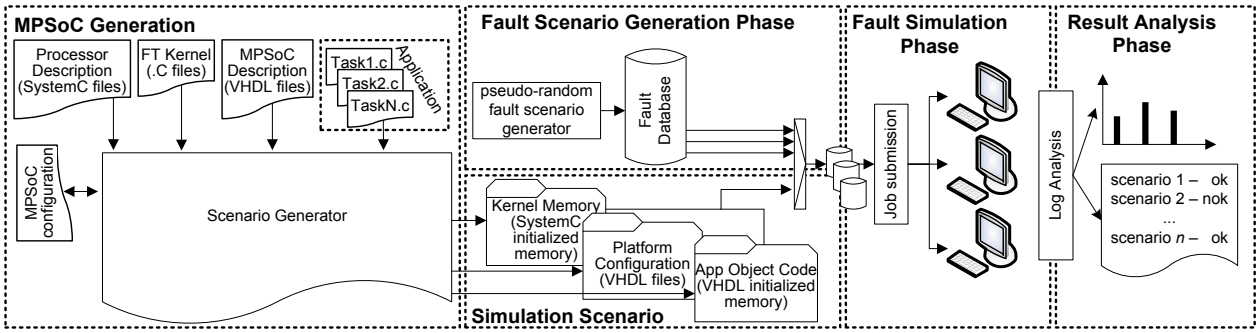


Figure 42 – Fault simulation flow.

The first phase, ***MPSoC Generation***, is responsible for generating and compiling the hardware and software of the MPSoC. As input, this phase receives the MPSoC hardware description, the FT kernel, the application code and the MPSoC configuration file. The configuration file contains parameters as MPSoC size, master processor location, local memory size and abstraction level of hardware description. In our experiments, the processor and local memory are described in cycle-accurate SystemC and the other modules are described in RTL VHDL, enabling smaller simulation time when compared with the full VHDL description [PET12]. The MPSoC *Generation* phase generates the ***Simulation Scenario***, which contains the compiled kernel, the compiled application, and the MPSoC hardware model.

The ***fault scenario generation phase*** generates a database of faults scenarios to be evaluated. These fault scenarios can be described manually, automatically, or a combination of both. Manual scenarios are used to describe special fault scenarios with corner cases. Automatic scenarios use pseudo-random generators to quickly generate hundreds of fault scenarios.

The ***fault simulation phase*** executes a fault simulation for each scenario in the database. A grid computing resource distributes the simulation jobs in parallel among workstations. Each simulation generates log files with the results, such as protocol latency, application execution time, among other informations. These log files are parsed in the ***result analysis phase***, extracting the performance information and checking whether the application was able to execute with faults. This phase also generates regression reports, charts, and tables used to compare each fault scenario.

Four applications are used in this evaluation. Two of them are synthetic applications (called *basic* and *synthetic* applications), used mainly for validation purposes, and two real applications (with actual computation) that implements part of an *MPEG* encoder the DTW (dynamic time warping) algorithm. These applications are decomposed in communicating tasks, illustrated in Figure 43. This figure also shows the task mapping into the MPSoC.

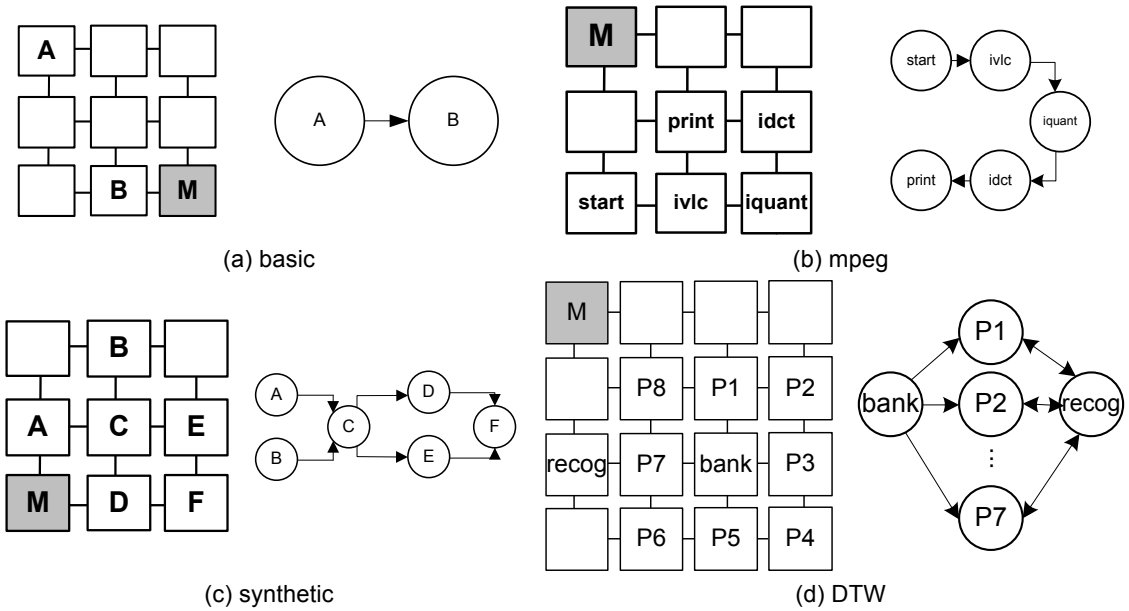


Figure 43 – Evaluated applications, their task mappings and task graphs.

Table 6 summarizes the validation process for single fault scenarios. Without comparing the approaches, we present some results that are common to the three methods:

- ***scenarios***: number of simulated scenarios;
- ***executed scenarios (%)***: the percentage of possible scenarios. 100 means that all possible fault scenarios were simulated;
- ***affected scenarios***: number of scenarios affected by faults. In this context, affected means that at least one task fires a seek request;
- ***node isolation***: the number of fault scenarios that caused system failure due to a node isolation.

Table 6 – Validation results with one fault.

	basic	synth	MPEG	DTW
scenarios	48	48	48	96
executed scenarios (%)	100	100	100	100
affected scenarios	8	12	8	91
node isolation	0	0	0	0

Results in Table 6 show that at least 8 fault injections affected the application execution (*affected scenarios* row), and the proposed approach was able to find an alternative path, enabling the application to finish its execution (*node isolation* row).

Table 7 details similar information, but assuming two faults per scenario. In this case, it is not possible to simulate all possible fault scenarios since it would take several days of CPU time. This way, 20% of the possible fault scenarios were randomly chosen for simulation.

Table 7 – Validation results with two faults.

	basic	synth	MPEG	DTW
scenarios	1213	1193	1213	2890
executed scenarios (%)	20	20	20	20
affected scenarios	215	394	251	1069
node isolation	2	3	3	10

The results in Table 7 show an increased number of affected fault scenarios (*affected-scenarios* row) but, for most of them, the proposed approach was able to determine an alternative path. However, there were some system failures (*node-isolation* row). The reason is that tasks might be isolated by faults, as illustrated in Figure 44. In this example, task A cannot send packets to task B because both outgoing ports (east and south) are faulty. Redundant hardware or task migration may restore the system execution when this scenario happens.

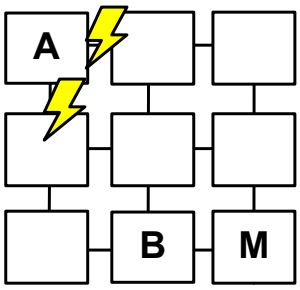


Figure 44 – Task isolation due to faults at east and south ports.

5.5.1 Performance Evaluation of Adaptive Watchdog Timers Implementation [WAC14a]

This section discusses how the faults might affect the Application Execution Time (AET) using watchdog timers (Section 5.4.1). The AET includes computation and communication time, and faults might affect the communication time by increasing the *protocol latency* (PL_f). In the case of watchdog timers, the PL_f is related to *k**AVG, the *time to find an alternative path* (TFAP), and the time to retransmit the packet.

The TFAP includes the time required to execute the seek, backtrack, and path computation steps, as presented in Section 4.4.2. The TFAP grows linearly with the number of hops between the source to the target tasks. Each hop in the path requires in average 30 clock cycles for the *seek* and the *backtrack* steps (15 clock cycles for each step). These steps are not time-consuming because they are implemented in hardware. The *path computation* step takes about 200 clock cycles per hop, and it is executed in software (kernel). For instance, the TFAP for a 10-hop path is approximately 2,300 clock cycles, being 300 clock cycles consumed by *seek* and the *backtrack* steps, and 2,000 clock cycles (200*10) to compute the path. The clock period in these experiments is 10 ns. Thus, 2,300 clock cycles correspond to 23 us while the AET is in the order of milliseconds.

The other relevant parameter for the AET is the time to fire the seek process, determined by *k**AVG. As discussed, the *average protocol latency* (AVG) is application-dependent, affecting in the *k* value.

To evaluate the impact of *k**AVG, the experiments presented in this section assumes a fixed task mapping for each application (Figure 43), and a fixed fault scenario (i.e. fixed fault location and fixed fault injection time). In this way, only the impact of *k* in the AET is evaluated. We start with a *k* equal to 12, decreasing it until false seek request starts. Just a few fault scenarios are presented in this section for the sake of clarity. In fact, although several scenarios inject fault at different locations, they present the same impact in the AET.

Figure 45 presents the AET for the *basic* application considering four fault scenarios (labels 0 to 3), single fault per scenario, and *k* = {2, 4, 8, 12}. With the increase of *k* value, the AET under fault condition increases up to 50%. Note that fault scenario also have a significant impact on the variability of AET, meaning that the fault location and time, which are not under control of the designer, have a relevant impact on AET in this method (watchdog timers). Scenario 0 (blue line) represents the fault free AET used as reference. Note that the proposed method affects the fault-

free AET only when *k* = 2, because some false seek processes were fired, slightly increasing the AET.

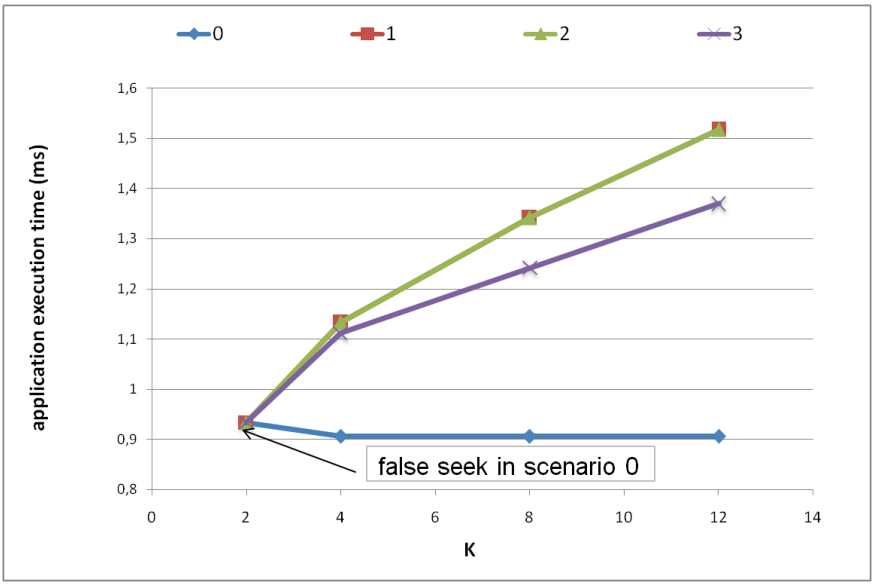


Figure 45 – Application execution time (AET) for basic application in scenarios with faults as a function of parameter k. Each line represents a given scenario with one fault.

Figure 46 shows the AET for the MPEG application considering three fault scenarios (labels 0 to 2), single fault per scenario, and k = {2, 4, 8, 12}. The MPEG AET with faults is less than 11% higher than the normal fault-free AET. One can observe that *k* has a smaller influence on AET compared to the *basic* application. In addition, scenario 2 with *k* = 2 present a slight increase in AET. This is caused by false seeks caused by the low k value, increasing the AET in the presence of a fault. The scenario 0 (blue line) is the fault-free reference.

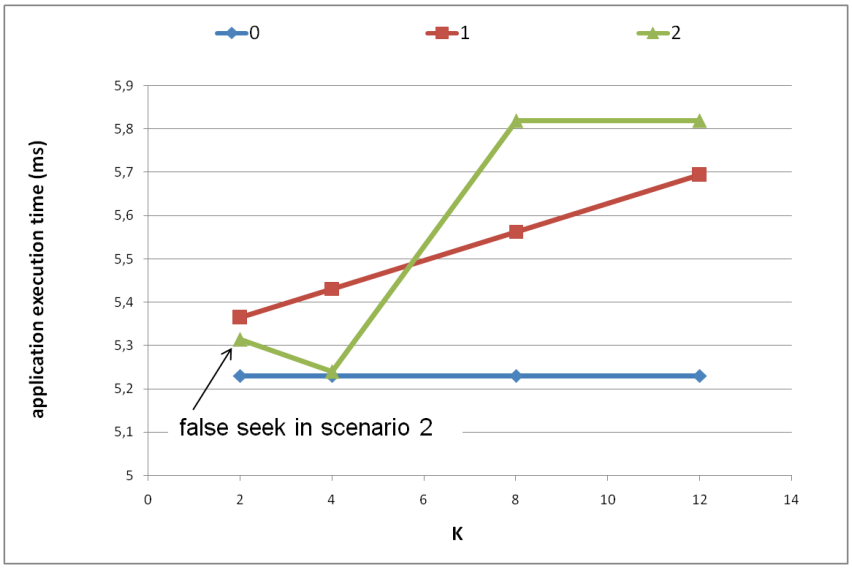


Figure 46 – Application execution time (AET) for MPEG application in scenarios with faults as a function of parameter k. Each line represents a given scenario with one fault.

The simulation of scenarios with *k*< 8 in the previous experiments fired a large number of false seeks. For this reason, Figure 47 shows the AET for the *synthetic* application considering six fault scenarios (labels 0 to 5), single faults per scenario, and k = {8, 12}. The *synthetic* AET with faults is about 13% higher than the normal fault-free AET. One can observe that *k* has a small influence on AET. The fault scenario (fault location and fault injection time) has more impact than *k* on the *synthetic* AET.

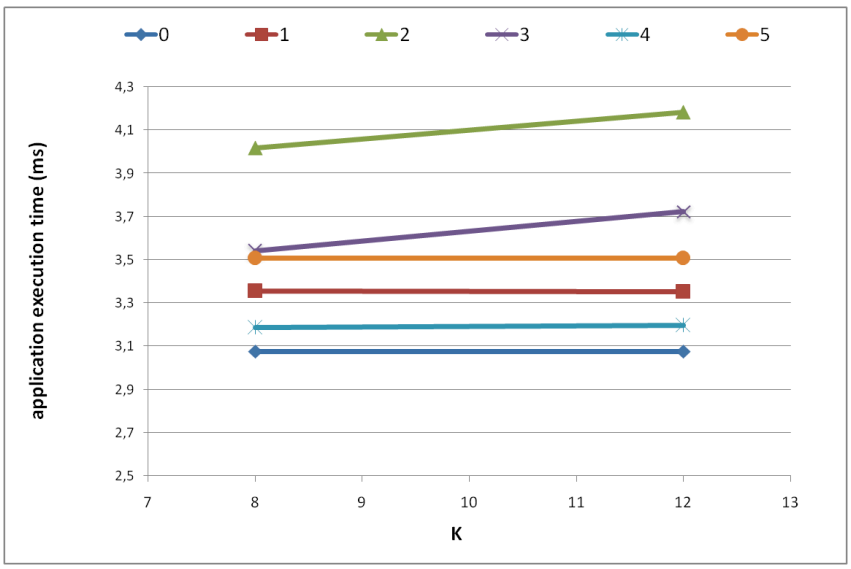


Figure 47 – Application execution time (AET) for synthetic application in scenarios with faults as a function of parameter k. Each line represents a given scenario with one fault.

In conclusion, these experiments show that the variation of AET under fault situations may be acceptable (~10%) when the computation time is higher than the communication time, as in real benchmarks (*MPEG* example). Applications with more average latency (AVG) variability requires larger *k*, therefore they might suffer larger AET variability under fault conditions.

The proposed method can only affect the AET under normal condition (fault-free) if *k* is too small for the application, due to the generation of false *seeks*. A simple communication application profile can help to determine the appropriate value of *k*.

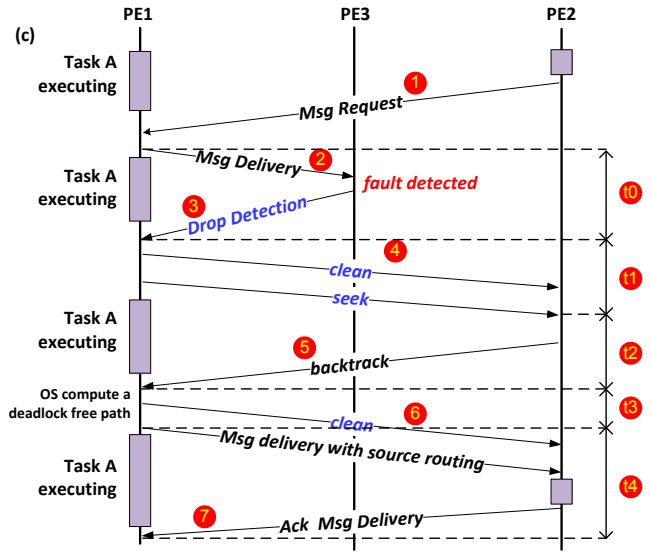
5.5.2 Performance Evaluation of Auto-detection of faulty paths [WAC14b]

Table 8 presents the performance of the FT communication protocol for the scenario presented in Figure 38, where a router has a single fault in the west input port, varying the distance between the source processor (PE_1) to the target processor (PE_2). The adopted benchmark is the MPEG, with 128-word messages (256-flit packets) transmitted from PE_1 to PE_2. There is no congestion in the path since the goal is to characterize the protocol. In Table 8:

- *t0* – time required to inject a packet into the network, and the reception of the fault detection. To inject a packet into the network, the OS: (*i*) configures the DMA module to transmit the packet; (*ii*) restores the context of the current task; (*iii*) enables external interrupts to be received. During this process, the fault detection interruption is received. As the time spent in the NoC is smaller than the number of cycles consumed by the OS, this parcel of the protocol is constant regardless the number of hops.
- *t1* – time spent between the fault detection by PE_1 and the reception of the *SEEK* by PE_2. The fault detection packet interrupts the OS if PE_1, triggers a clean wave to empty the fault detection, and then triggers a seek to PE_2. It increases, in average, 16 clock cycles per hop.
- *t2* – time to transmit the backtrack packet. Due to the hardware implementation, a small number of clock cycles is observed, and it is a function of the number of hops.
- *t3* – corresponds to the number of cycles to compute the new path in the OS. It is also a function of the number of hops.
- *t4* – time spent to deliver the packet using the faulty free path. The number of clock cycles is constant for the same reason explained in the parcel t2→t3.

Table 8 – Number of clock cycles for each step of the FT communication protocol of Figure 39 varying the number of hops between PE_1 and PE_2.

hops #	t0	t1	t2	t3	t4	TOTAL
2	769	647	48	981	1023	**3468**
3	769	672	51	1084	1023	**3599**
4	769	688	63	1177	1023	**3720**
5	769	704	75	1291	1023	**3862**
6	769	720	87	1402	1023	**4001**
7	769	736	99	1499	1023	**4126**



In these scenarios, the amount of time to transmit a packet in a fault-free scenario is in average 1670 clock cycles (Figure 48). Therefore, the worst-case overhead is observed in the 7-hops scenario, corresponding to 2.46 times the time to transmit a message without fault. The transmission of the first message (Figure 48) is faster because it is buffered while the consumer task starts its execution. The third message requires path reconfiguration and message retransmission. All others messages (4 to 8) are not penalized since the new path is also minimal, and the time to transmit the message in both cases is the same. Consequently, the impact of the fault-detection/path reconfiguration/retransmission in the application is small, since the process is executed once per fault.

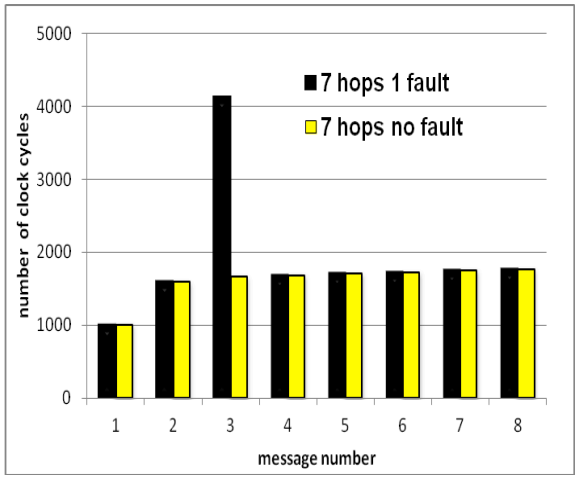


Figure 48 – Time spent to transmit eight 256-flit packets with and without fault using the proposed FT communication protocol for the 8 first frames. The fault was detected in the third packet.

For the Auto-detection of faulty paths, three applications were evaluated with 1,200 scenarios. Two faults are injected at each scenario at the same moment, at random ports. The simulations demonstrate that the method supports several simultaneous faulty ports, as long as the number of faults does not physically split the application graph into disjoint graphs. Figure 49 presents the execution time for the 3,600 simulated scenarios.

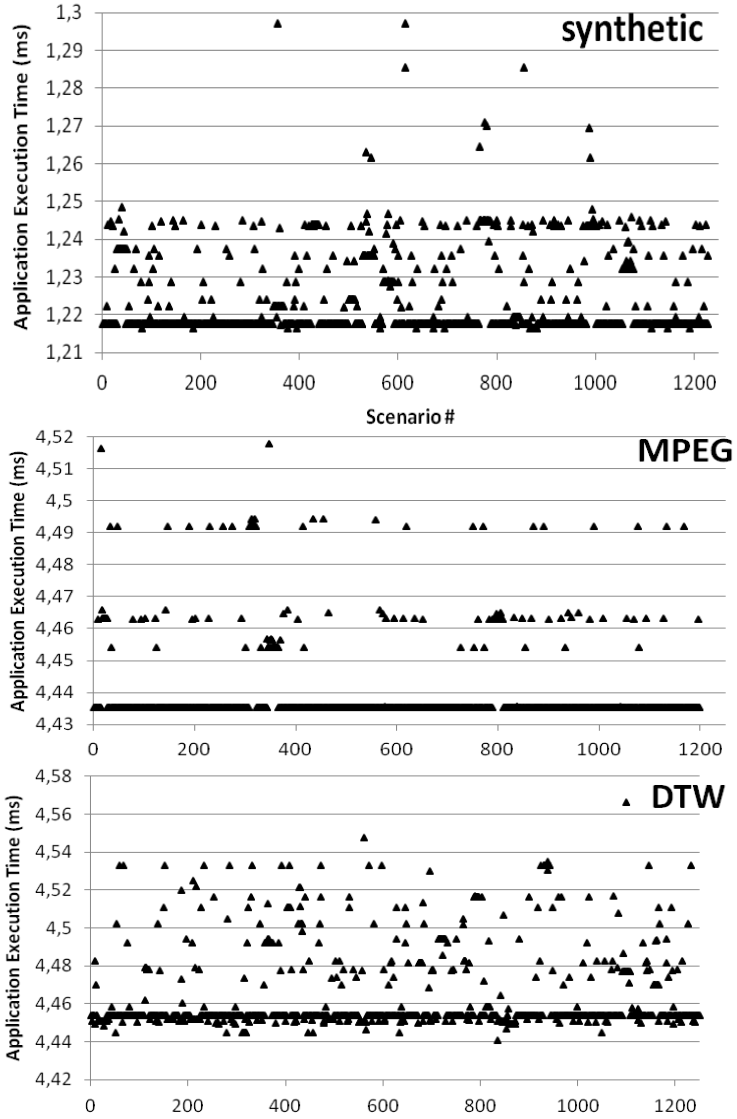


Figure 49 – Applications execution time for each faulty scenario. Each dot in the figure represents a given scenario.

Each application execution length is such that one message (out of 10) is affected by a fault, ensuring a pessimistic evaluation scenario where the fault impact is not dissolved in an extremely long application execution time. The worst-case execution time overhead was 2.53%, 1.87%, and 6.5% for the DTW, MPEG, and synthetic applications, respectively. The average-case overhead was 0.17%, 0.09%, and 0.42%. Three reasons explain this remarkable result, compared to the previous approach:

- Independence of the fault location. The overhead is mostly induced by the moment of the fault detection. If the PE is executing some OS function, the treatment of the interruption due to the fault may be delayed. In other words, the time spent to treat the interruption is much longer than the time to receive the fault notification.
- Fast path reconfiguration. The retransmission of a given message roughly double the packet transmission time (see Figure 48), with a small overhead due the fast path reconfiguration (Table 8).
- Overhead only in the undelivered message. After path reconfiguration, there is no overhead in the communication since the new path is saved in the PE for future transmission.

5.5.3 Performance Evaluation of Auto-detection of faulty paths with simplified protocol

The results for the Auto-detection of faulty paths with simplified protocol showed a decrease of the AET compared to the previous approaches. The AET reduction comes from the simplification of the communication protocol that removed the acknowledgement message. Results concerning this implementation are detailed in the next section (Table 9).

5.5.4 Overall Evaluation of the FT Communication Protocols

Table 9 presents the AET worst-case for the three methods, with 1 and 2 simultaneous faults. The DTW application was removed in the *watchdog timer* implementation. The reason is the variability in the average latency, since multiple tasks communicate with the same task (e. g. *recog* task in Figure 43), generating too many *false seek* requests.

Table 9 summarizes all the executed scenarios presenting the comparison for AET worst-case overhead for the three methods. The proposed methods were able to treat the injected faults. The first approach showed the higher overhead due to the large delay to trigger the path search method. The two other approaches showed a small impact in the AET when compared to watchdog timer. Table 9 also shows that multiple simultaneous faults do not necessarily increase the worst-case AET since the seek request associated to each fault is executed in parallel.

Table 9 – AET worst-case overhead for Watchdog timers and the two Auto-detection of Faulty Paths approaches.

Approach	Watchdog Timers		Auto-detection of Faulty Paths		Auto-detection of Faulty Paths with simplified protocol	
	number of fault(s)		number of fault(s)		number of fault(s)	
Application	**1**	**2**	**1**	**2**	**1**	**2**
basic	67.46%	67.46%	N/A	N/A	7.46%	5.85%
mpeg	68.47%	68.55%	2.16%	1.87%	0.66%	0.43%
synthetic	35.99%	38.79%	7.5%	6.5%	2.38%	1.45%
DTW	N/A	N/A	3.46%	2.53%	0.79%	0.60%

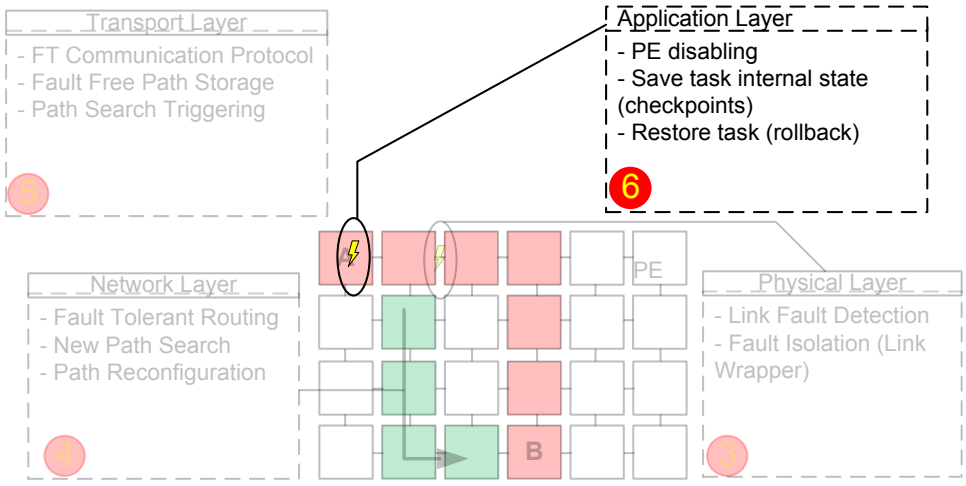
The hardware overhead due to the FT protocols comes from the drop_detection, with a total cost 248/270 LUTs/FFs respectively (there is 8 of these modules in the router). The memory usage overhead for the kernel with the proposed FT is 6.5 KB. For comparison, the memory usage overhead in [HEB11] is 8.1 KB. Such values may be considered small for embedded systems.

5.6 Final Remarks

This Chapter presented a FT communication protocol for NoC-based MPSoCs. Both hardware and software were integrated and validated on an existing MPSoC design described at the RTL level. The proposed method was evaluated with synthetic and real applications while permanent faults were injected into the NoC at runtime. The protocol automatically detects the unreachable tasks and launches the search for a faulty-free path to target PE. This way, the method enables applications to continue their execution, as long as there is at least a single functional path to the target PE. The entire process of finding alternative paths takes typically less than 2000 clock cycles or 20us. The protocol caused less than 1% in the total execution time of computationally intensive applications in case of faults.

Note that the last modification, removing the acknowledgment message is valid only for the communicating messages (message request and delivery). Therefore, other messages exchanged with the MP, for example, should include acknowledgment messages to provide FT.

6 APPLICATION RECOVERY



This Chapter presents the fourth and last layer of the Thesis: the application layer. This layer proposes a lightweight error recovery technique for multi-core systems.

This platform used to validate the methods presented in this Chapter is different from the platform used in the previous layers. The implementation of this layer occurred during the Ph.D. internship at the CEA-List laboratory, under the supervision of the Dr. Nicolas Ventroux. This architecture, named P2012 [BEN12], is a multiprocessor platform, using shared memory as communication model.

The goal of this layer is to execute a checkpoint method periodically, and then if a fault is detected, the previously saved context is restored, allowing the system to continue its execution with unaltered data.

6.1 The State-of-the-Art in Context Saving and Recovery Techniques

This section reviews the state of the in context saving and recovery methods. There are common approaches targeting HPC and some for embedded systems.

6.1.1 ReVive [PRV02]

The Authors in [PRV02] present ReVive, a checkpoint/rollback mechanism for architectures with processors, caches and memory interconnected by an off-chip network (e. g. HPC). They implemented a partial separation technique with a logging checkpoint mechanism. This approach proposes a partial separation, where checkpoint data and working data are one and the same, except for those elements that have been modified since the last checkpoint.

The memory is modified in the following manner. When a write-back arrives, the method checks whether this is the first modification of the line since the last checkpoint. If it is, the previous content of the line is read from memory and saved in the log before the new content of the line is written to memory. A modified line only needs to be logged once between a pair of checkpoints. To this end, the directory controller is extended with one additional state bit for each memory line, which the Authors call the Logged (L) bit.

This approach requires the memory to be divided into pages, with a hardware directory controller responsible for the access to the memory.

For the evaluated applications, the approach imposes a worst-case area overhead of 2.5 MB for storing the checkpoint data. The results on a 16-processor system indicate that the average error-free execution time overhead of using ReVive is only 6.3%.

6.1.2 Chip-level Redundant Threading [GON08]

In [GON08] the Authors propose a Chip-level Redundant Threading (CRT) to detect transient faults on Chip Multiprocessors (CMPs). The approach is to execute two copies of a given program on distinct cores and then compare the stored data. CRTR (CRT with Recovery) achieves fault recovery by comparing the result of every instruction before commit. Once detecting different results, the microprocessor could be recovered by re-executing from the wrong instruction. The results showed that the performance overhead of the context saving when compared to the baseline processor is approximately 30%.

6.1.3 Reli [LI12]

The Reli technique [LI12] proposes to change the micro-operations of instructions, which stores registers and data memory. At runtime, Reli instructions execute not only native functionalities (e.g., adding two operands of the ADD instruction), but also Reli functionality (e.g., generating checkpoint data of destination register for ADD instruction). They modify the micro-operations in the instructions that change the state of the processor.

They adopted two stacks, named *backup stacks*, used for storing the registers in the register file and for storing the data memory values that changed. For registers, the first time a register is changed in a basic block, the old value of that register is stored in the backup stack. For data memory, the old values are simply stored in the data memory stack along with the address of the location.

After profiling (with SPEC INT 2006 and MiBench suites), the Authors concluded that the maximum number of stack positions necessary was 66. Results show an overhead of 1.45% in average execution time and 2.4% worst-case on a faulty-free scenario.

For fault injection, the Authors used a single bit-flip as the fault model and the number of fault injections was 1000 for each application. They injected one fault for each application iteration. For the evaluated applications, the worst-case recovery overhead is 62 clock cycles and the highest application average recovery time is 17.9 clock cycles.

Regarding the cost of the proposal, the area overhead varies from 37.8% to 52.0% for the evaluated applications (Reli implementations differ according to the application) compared to the baseline implementation and 9.6% to 52.4% more leakage power.

6.1.4 DeSyRe [SOU13]

The DeSyRe project [SOU13] presents an MPSoC framework for FT purposes. The Authors investigate techniques for checkpointing/migrating task state at each PE of the DeSyRe SoC. The Authors claim that initial results indicate that maintaining duplicate copies across the NoC offers a superior performance, but do not present this result. Furthermore, the duplicate copies have the added benefit of protecting against NoC failures and disconnected nodes. According to these initial results, the project assumes an architecture with a main memory being fault-free while the PEs and local memories are fault-prone.

As error recovery technique, they propose the checkpoint and task re-execution. They evaluated the MPSoC mapped to a Virtex6-based ML605 development board. The MPSoC consists of a master Microblaze to host the runtime system, and 7 workers for tasks execution.

The evaluation is conducted with a matrix multiplication application with 24 tasks, varying the matrix size. In addition, they consider three fault error rates: 4%, 20% and 41%. As this is an ongoing work the Authors do not evaluate the checkpoint technique, therefore there is no results related to the overhead in a fault-free scenario. However, the evaluation of the application re-execution in the scenario with 20% of tasks being faulty, the execution time doubles.

6.1.5 Barreto's Approach [BAR15]

Barreto et al. [BAR15] propose an online fault recovery for embedded processors of MPSoCs based on distributed memory. This approach automatically restarts affected applications reallocating tasks to healthy processors. All steps are performed at the kernel level, without changing user application code. The Authors employ HeMPS as the target platform.

The process is described as follows:

(1) The MP (Manager PE) receives a fault notification message, indicating a fault at a SP (Slave PE).

(2) The MP constructs a set TF = $\{t_1, t_2, \ldots, t_n\}$ representing the affected tasks running in the defective SP.

(3) The MP identifies for each t_i in TF the applications affected the fault, constructing a new set TA = $\{A_1, A_2, \ldots, A_m\}$, being A_i an application to be frozen.

(4) The MP sends a freeze message for all tasks of each application in TA.

(5) The MP relocates only the tasks in TF to healthy SPs. All other unaffected tasks stay in the same location. After the dynamic task mapping, the unaffected tasks receive the new addresses for the relocated tasks.

(6) MP sends the relocation and the unfreeze messages for all tasks of each application in TA.

The results shown the recovery overhead compared to the application execution time is 1,8% for one fault and for seven faults 5%. The recovery overhead is impacted only by the reallocation time. In this proposal only the tasks affected by faults are reallocated, reducing the overhead. Regarding the area overhead, there is no impact since all modifications were in the software. There is no information for the kernel memory footprint overhead.

In fact, the Authors present a solution for application restarting. When a fault is detected in a given PE, the application is restarted. Note that the application restarting is not a problem for periodic applications, like video processing, representing the lost of few frames, in practice imperceptible for the final user. However, the proposed approach cannot be applied for application requiring the past state, as scientific computations.

6.1.6 Rusu's Approach [RUS08]

The Authors in [RUS08] present a checkpointing method for NoC-based MPSoCs with message passing. The proposal is a coordinated checkpoint, where a checkpoint initiator (a dedicated task) asks to all tasks to perform checkpoints. As soon as possible after receiving this request, a task takes a checkpoint, then sends an acknowledge message to the initiator. When the initiator knows that all the tasks took their checkpoints, the new global checkpoint is validated and the checkpointing procedure finishes.

As the initiator should send broadcast messages to all non-initiators, the Authors noticed that it was inducing congestion and consequently increasing checkpointing latency. Then it is

proposed an optimization in the protocol (changing the complexity from $O(n^2)$ to $O(n)$) and in the network with a more efficient broadcast method.

The simulations execute a single application with a unique task mapped to each PE. Each task generates a uniform traffic with a constant rate of message injection for every task (0.005/cycle) and a constant message length (64 bytes).

Results show that, with the modification in the protocol and in the broadcast, the overhead of executing a checkpoint can be reduced to up to one order of magnitude in systems with more than hundreds of PEs. The Authors evaluate the average checkpoint log size for each implementation but do not discuss the size for a real application and how the dedicated memory to the log can be estimated.

6.1.7 State of the art discussion

Table 10 presents a comparison of the evaluated works. Some of the methods shown an overhead without faults smaller than 20%, considered by the Authors an acceptable overhead [PRV02][LI12]. However, these approaches target distributed systems [PRV02] or require modification in the ISA and dedicated hardware [LI12]. Basically all shared memory context saving techniques are based on this two works.

In architectures with message passing, the largest problem is the overhead to synchronize a global state for messages in each PEs [RUS08]. [BAR15] et al. adopt a simpler approach, since the proposal do not present a global synchronization mechanism, restarting the whole application in case of faults.

Other methods present a large area overhead and require redundant executions, wasting processing resources [GON08]. In [SOU13], the Authors present an ongoing work, with no context saving, just an application restarting approach.

Table 10 – Comparison of evaluated Fault-Tolerant Context Saving approaches.

Proposal	References	Target Architecture	Communication Model	Area Overhead	Fault free Performance Overhead	Recovery Overhead
ReVive	[PRV02]	HPC	shared memory	2.5MB memory	6.3%	59ms worst case/17ms avg
CRTR	[GON08]	CMPs	shared memory	100%	30%	19% when compared to the fault free execution
Reli	[LI12]	embedded single-processor	N/A	79.3% worst case	2.4% worst case/ 1.45% avg	62 clock cycles worst case 17.9 clock cycles avg
DeSyRe	[SOU13]	MPSoCs	N/A	N/A	N/A	doubles the execution time for 20% of tasks being faulty
Barreto's approach	[BAR15]	MPSoCs	message passing	zero (software approach)	zero	5% worst case
Rusu's approach	[RUS08]	MPSoCs	message passing	N/A	N/A	N/A
Proposed Work		MPSoCs	shared memory	external - L3 to store context	5.67%	17.33% - 28.34% (1 – 3 faults)

6.2 Fault-tolerant Reference Platform

The P2012 MPSoC architecture is an area- and power-efficient many-core architecture for next-generation data-intensive embedded applications such as multi-modal sensor fusion, image processing, or mobile augmented reality [BEN12][MEL12]. The P2012 contains multiple processor clusters implemented with independent power and clock domains, enabling fine-grained power, reliability and variability management. P2012 can reach 19 GOPS (with full floating point support) in 3.8mm² of silicon with 0.5 W power consumption.

6.2.1 Architecture

Figure 50 presents an overview of the P2012 architecture. P2012 is a GALS fabric of tiles, called clusters, connected through an asynchronous global NoC (GANOC) [THO10]. Each cluster has access to an L2-shared memory and to an external L3-shared memory. Each P2012 cluster aggregates a multi-core computing engine called ENCore and a cluster controller (CC).

The ENCore contains 16 STxP70-V4 processing elements (PEs). The STxP7 70-V4 has a 32-bit load/store architecture with a variable-length instruction-set encoding (16, 32 or 48-bit) for minimizing the code footprint. Instructions can manipulate 32-bit, 16-bit or 8-bit data words, as well as small vectors of 8 bits and 16 bits elements. The STxP70-V4 core is implemented with a 7-stage pipeline for reaching 600MHz and it can execute up to two instructions per clock cycle (dual issue).

Within the cluster, one STxP70-4 processor is implemented with a floating-point unit extension (FPx). In this configuration, one single cluster can execute up to 16 floating point operations per cycle. In addition, since all cores have independent instruction issue pipelines, there is no single-instruction, multiple-data restriction on execution, which is a common restriction of GPUs. This greatly simplifies application development and optimization.

Each core has an L1-16KB-private instruction cache. For the data cache, the PEs share an L1-256KB tightly coupled data memory distributed in 32 banks (TCDM). This way, the PEs does not have private data caches or memories, therefore avoiding memory coherency overhead.

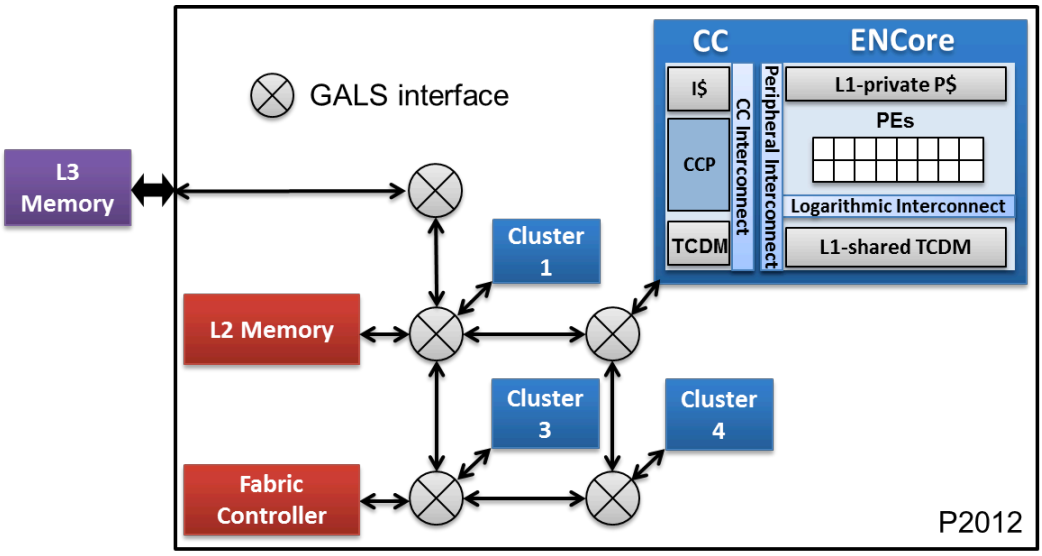


Figure 50 – P2012 architecture.

The CC is the manager processor of its cluster. The CC encloses a processor named CCP (Cluster Controller Processor), a DMA subsystem and two interfaces: one to the ENCore, one to the GANOC. The CC processor also adopts the STxP70-V4 processor, 16-KB of program cache and 32-KB of local data memory. The cluster controller processor, together with its peripherals, is in charge of booting and initializing the ENCore. It also performs application deployment on the ENCore. The DMA sub-system has two independent DMA channels. It performs the data block transfers from the external memory to the internal memory and vice-versa while the various cores are operating. The CC interconnects supports intra and inter-cluster communication.

6.2.2 Software Stack

The software stack is named HARS [LHU14] and it is based on a hardware-assisted runtime software. It is composed of resource management features, multiple execution engines to

support different programming models, and synchronization primitives relying on a hardware module named *HardWare Synchronizer* (HWS). ENCore provides scheduling and synchronization acceleration by means of the HWS. The HWS includes a synchronization module that provides a hardware-supported acceleration of various synchronization mechanisms: semaphores, mutexes, barriers, joins, etc.

The HWS is dedicated to accelerate synchronization primitives on massively parallel embedded architectures. It is designed as a peripheral to be integrated in architectures using load/store operations, providing their runtime software with efficient synchronization implementations even for architectures without atomic operations support. It can also remove polling issues related to spin-lock operations. A specific set of software synchronization primitives based on this hardware accelerator can be used by the different cores to perform synchronizations. Thus, instead of using a software instruction requiring an atomic memory read/write access, the synchronization primitives leverage the HWS atomic counters to implement locking. Moreover, the runtime software uses sleep locks to put the processor in a waiting state until it is awakened when the resource is free.

HARS proposes a small set of execution engines covering a wide range of parallel programming styles. Two main execution engines are implemented: conventional multi-threading for coarse grain parallel expression (suitable for thread-level or task-level parallelism) and synchronous and asynchronous reactive tasks management for fine grain parallelism (suitable for data-level parallelism).

Finally, an API enables the software designer to have access to all communication primitives, parallel task execution triggering and control of the synchronization features presented in other layers. The Software Stack is important as all the primitives of the saving and recovery context are implemented in these layers.

6.2.3 Execution model

In the P2012, a conventional multi-threading execution model based on fork-join mechanisms has been chosen. The PE that executes the fork is referred as master PE (PE_m). Any of the 16 PEs of the ENCore may be select as PE_m. As showed in Figure 51, PE_m executes the sequential part of the application and can delegate tasks to other processors, parallelizing the execution.

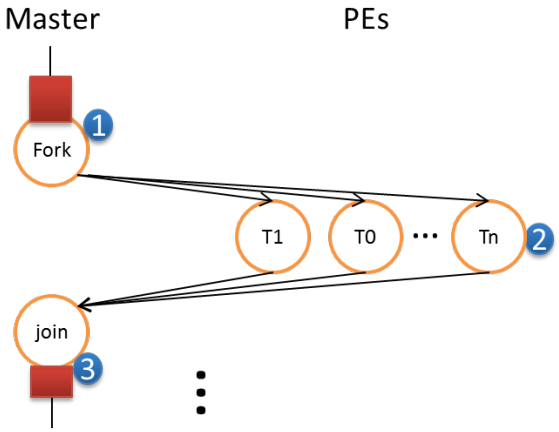


Figure 51 – Execution Model in P2012 with HARS. (1) master forks parallel tasks, (2) other PEs execute the tasks, and (3) the master does the join.

PEs only executes tasks that were forked by PE_m. To execute the fork, the PE_m populates a table *T* in a shared space with tasks to be executed. The fork procedure loads the local shared

memory within the cluster with the data and instructions to be executed by the parallel tasks. After the load procedure is executed, each PE runs its task. When there are no more tasks to be scheduled, the PE_m waits until all tasks have finished their execution to join the tasks. Every PE that is not doing a fork operation executes a scheduling loop. This loop searches for jobs to be executed by scheduling ready tasks from table *T*.

Each PE accesses a dedicated shared memory space, which is released when the task finishes its execution. At this point, the local memory in the cluster accessed by the PE has no useful information for the execution on PE_m and can be discarded. The fork-join process can be repeatedly executed, but the PE_m must wait all tasks to finish their job before the join.

6.3 Proposed Fault-Tolerant Method [WAC15]

As stated at the beginning of Chapter, this Thesis proposes a fault-tolerant approach to tackle faults occurring in the processors. We assume that the fault detection scheme is out of the scope of our approach. Implementations have already been proposed, as discussed in section 2.3.

All the modifications are implemented at the Software Stack (HARS). The simulation and system evaluation is conducted with the model of the platform. This model is implemented at the transaction level modeling (TLM).

According to the execution model, only the context of PE_m is saved, as well as the global shared memory space. Thus, the context saving/restoring process is performed before the fork and after the join. This guarantees a coherent state for all PEs and eases the management of faults.

Thus, the execution context that must be considered is a structure composed by the 32 PE_m's registers, the .data section that stores all the shared uninitialized data, the .bss section that stores all the shared initialized data, and the PE_m stack which is locally stored in the CC L1-data memory.

This context structure is stored in the L3 memory, accessible by all the clusters. All accesses to this memory are made through the GANoC, inducing network traffic. The access time is higher when compared to the local shared memory within the cluster. The context structure is allocated at runtime according to the size the application needs.

For the results section, we modeled the access time for each memory. This time is taken into account only for the first access. The subsequent operations are executed in one clock cycle. Table 11 presents the access latencies.

Table 11 – Access time for memories in P2012.

Module	Access time (clock cycles)
READ/WRITE IN L1	1 CYCLES
READ/WRITE ACCESS L2	21 CYCLES
READ/WRITE ACCESS L3	200 CYCLES

The HARS software stack in P2012 does not allow the PEs to send an interruption to the PE_m. Then, it is not possible to interrupt the fork execution at the exact moment the fault is detected. The proposal is to use an atomic counter *faulty_PEs* to store the information if the PE is faulty or not. At the end of the parallel task execution, PE_m verifies if there was an error in some PE, reading its atomic counter. If a given PE is faulty, it is isolated from the execution processor list and consequently will not execute any other task. Then, PE_m starts the recover context procedure.

The fault insertion is executed by a dummy task that sets the *faulty_PEs* atomic counter if a fault is detected. Then the management of isolation is performed by the use of atomic counter, which is checked each time the scheduler function on each PE is called.

Figure 52 presents the proposed Fault-Tolerant execution model. Before the sequential execution is forked, the master saves the application context. This means that it stores in the global shared memory (in this order): (1) all processor registers; (2) its stack: (3) its .bss section and (4) its .data section. At the end of the fork/join process, the master checks if any of the PEs detected a fault and if needed, it triggers the recover context procedure. If there was no fault, the execution continues normally.

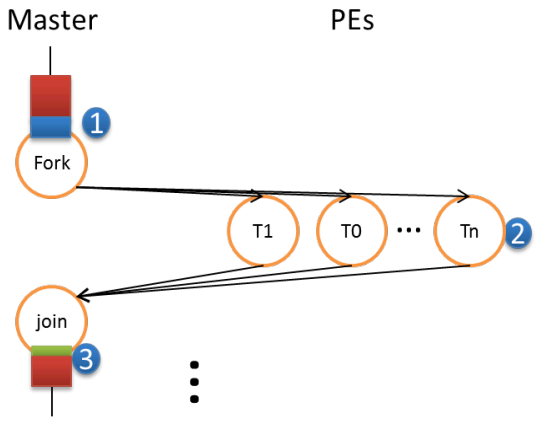


Figure 52 – Fault-Tolerant Execution Model: In (1) the master executes a context saving and in (3) it verify if there was a fault, if positive, the context is restored and the fork is re-executed avoiding the faulty PE.

6.4 Evaluation of the Proposed Fault-Tolerant Method

This section evaluates the overhead induced by the FT proposal. All scenarios are executed in the P2012's SDK released by ST Microelectronics. For the results, only 1 cluster is considered, and all the communications between tasks are made through the global shared memory space in L3 and memory accesses are made through GANOC.

Two applications are adopted as benchmarks. The first application is synthetic, with it task graph presented in Figure 53. A parameterizable number of NOP instructions (N) defines the task size. It is also possible to parameterize the number of tasks (T), and the number of iterations (R).

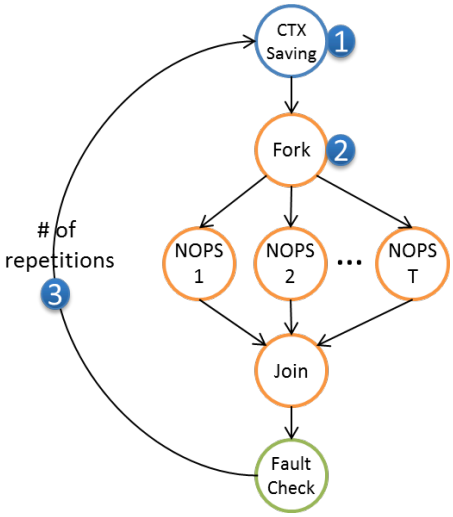


Figure 53 – Task Graph of the synthetic application. (1) The PE_m executes the context saving, (2) the fork splits the execution in *T* tasks, each one executing *N* number of *NOP* instructions, and (3) this process is replicated *R* times.

The second application is an industrial application named Human Body Detection and Counting (HBDC). It consists of processing an image sequence to determine the background image and subsequently the moving objects of the scene. The first phase uses the Mixture of Gaussian (MoG) technique [MCL00]. It is forked in 60 tasks, each one taking around 340,000 clock cycles to execute. Then, the remaining tasks are sequential. The moving objects are classified to determine whether they correspond to human shapes where 64 image frames are processed.

6.4.1 Evaluation of the method with Synthetic Application

Figure 54 presents the evaluation of the impact of the context saving varying the task size (*N*). The time to save the context is not a function of *N*. Thus, the size of the parallelized tasks should mask the context saving overhead. As shown in the Figure, the execution overhead reduces as N increases. A task with 10,000 NOPs has an overhead close to 20%, which is considered an acceptable overhead. The next experiments use N=10,000 as reference for the task size.

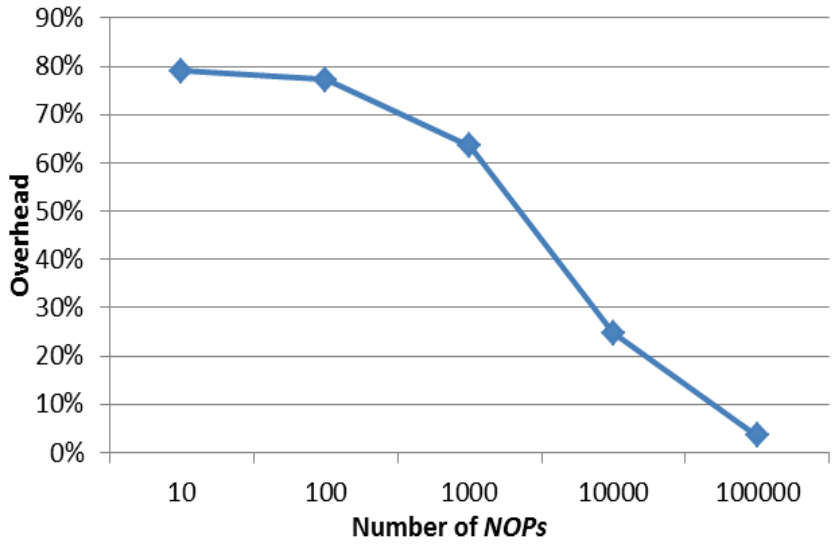


Figure 54 – Execution time overhead varying the number of NOPs in each task (T=10, R=10).

The next experiment evaluates the impact of the context saving, varying the size of the sections *.data* and *.bss* (Figure 55). The amount of data to save is the main limitation of the approach. A trade-off has to be defined between the tasks' execution time and the context data size. Then the programmer can choose an acceptable overhead cost of the context saving.

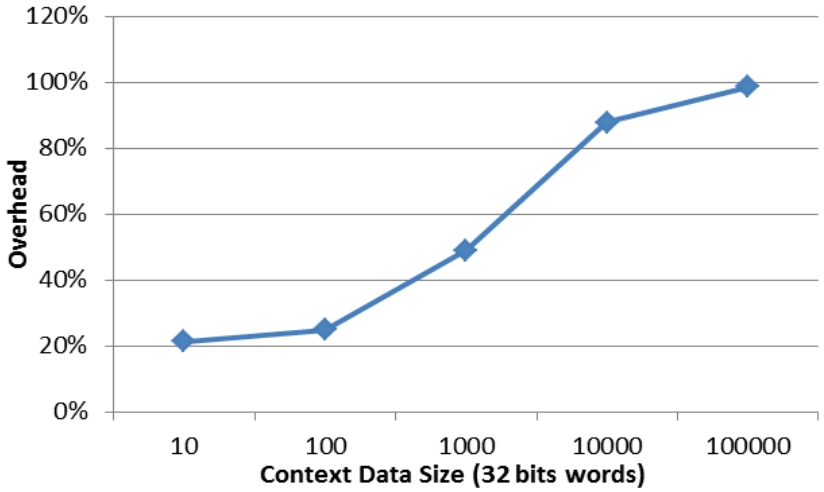


Figure 55 – Execution time overhead of context saving changing the context data size from 10 to 10k words of 32 bits (T=10, R=10, N=10,000).

The context saving is disabled to enable the evaluation of the fork overhead. Figure 56 shows the execution time overhead varying R. The Figures shows that the fork execution takes approximately 6% of the execution time. Since this overhead is independent of the number of repetitions, the fork/join process has a limited impact on the performance.

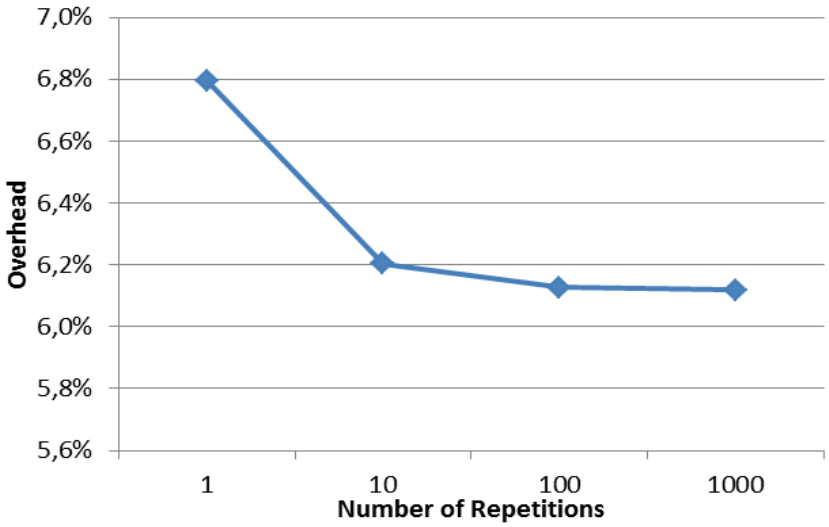


Figure 56 – Fork overhead varying the number of repetitions (T=10, N=10,000).

Figure 57 shows the execution time for the synthetic application without context saving (noFT), with the FT method and no injected fault (0faults), and with the FT method and a variable number of injected faults. The context saving implies a 35% overhead (N=10,000), and for each injected fault there is an increase of 15% for the context restoring and fork rescheduling.

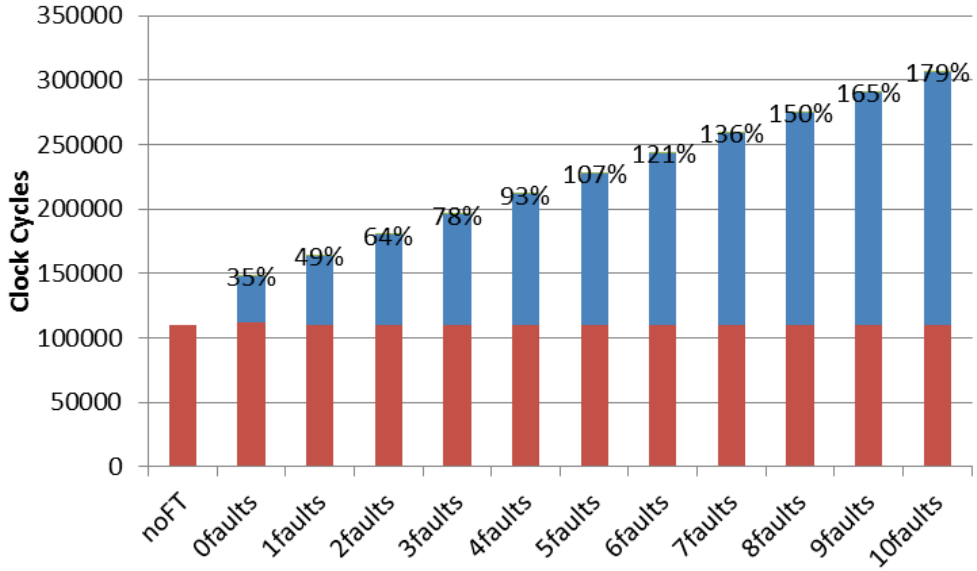


Figure 57 – Application execution time overhead for scenarios with no context saving, and the overhead for scenarios where there is overhead increasing the number of faults (*T*=10, *R*=10, *N*=10,000).

6.4.2 Evaluation of the method with HBDC Application

Figure 58 shows the execution time overhead when context saving is executed according to a variable number of frames. Saving the context at each eight frames increases the execution time

by 5,67%. This means that the background images will be restored as it was eight frames back if a fault is detected. For this application, the checkpointing frequency has only a QoS impact that will depend on the application frame rate. With a high frame rate, losing some frames will not affect the application, resulting in a good tradeoff between performance and quality.

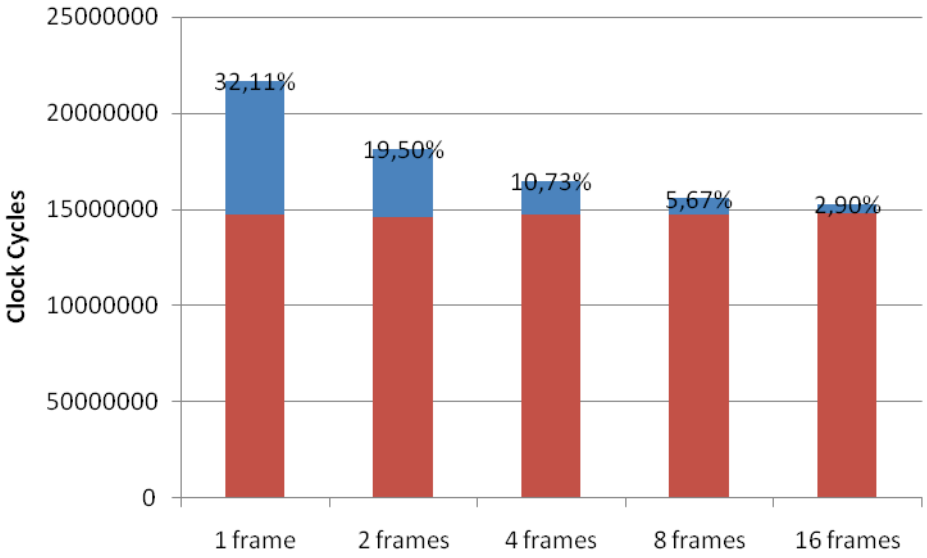


Figure 58 – Execution time overhead without faults when executing context saving from each frame to each 16 frames. The bars show the context saving overhead and the execution time.

Figure 59 presents five executions of the HBDC application, assuming context saving at each eight frames. In the first column (noFT), there is no context saving, being the baseline execution time. The second column shows the overhead induced by the context saving with no fault insertion (5.67% compared to baseline). The last three columns show the overhead for one, two and three faults in frames of different saved contexts of the application. Note that the percentage represents the overhead compared to the baseline and the highlighted part represents only the context restoring overhead. As there are tasks to be re-executed, the task execution time increases when the number of restored context grows.

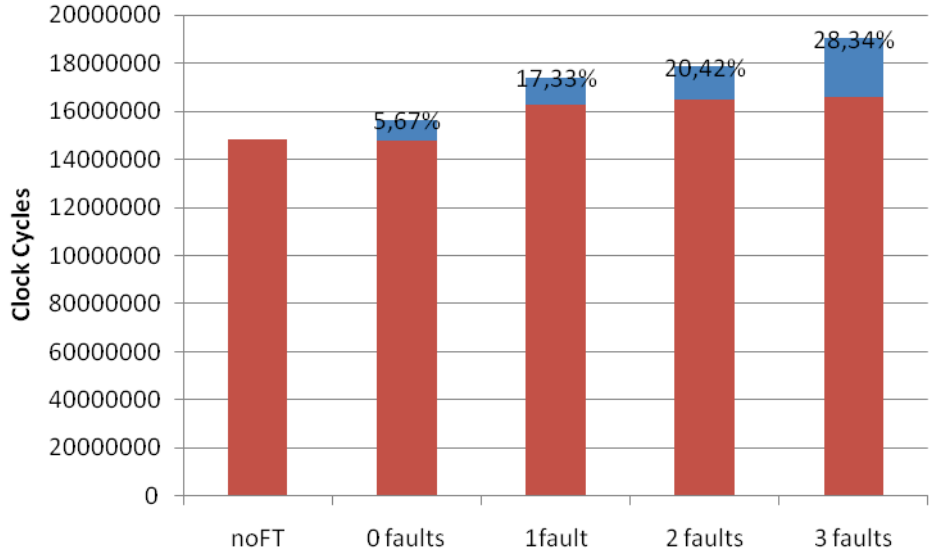


Figure 59 – Application execution time with no context saving, the overhead induced by the context saving and the overhead induced by the context saving plus the recovery time for one, two and three faults. The percentages represent the overhead compared to the baseline. The highlighted part represents the time executed restoring the context.

6.5 Final Remarks

This Chapter presented a Fault-Tolerant Context Saving for a state-of-the-art shared memory MPSoC. Results show that the proposal was validated and can recover applications from faults occurring in PEs. Execution with an industrial application shows a good tradeoff between execution time overhead with no faults (5.67%) and with faults (17.33% - 28.34%). The proposal does not imply in hardware overhead or redundant executions, as other works in the state-of-the-art.

The proposal presented a method for shared memory architectures, adapting existing techniques for distributed systems in embedded systems. The original contributions rely on (*i*) in the evaluation of the cost of the techniques in the P2012 architecture and (*ii*) an isolation technique to isolate a faulty internal core.

We adopt two assumptions in the current Chapter: (*i*) fault detection is out of the scope of this research; (*ii*) there are no pragmas or code added by the software designer, allowing context saving at any moment of the application execution.

A limitation of the proposed method is that it is very dependent to the communication model, shared memory. The speedup of the overall application is limited to the part where the application can be parallelized. Thus, our limitation is coupled to the fork-join approach.

The first step towards fault-tolerance for distributed memory architectures was presented in [BAR15]. A careful evaluation of context saving/rollback is an important work to be accomplished in a short term for the HeMPS platform. As stated in the state-of-the-art, this kind of communication model presents a synchronization problem, being necessary to have a centralized node to coordinate all the tasks to take their checkpoints.

7 CONCLUSION

From one side, MPSoCs provide a massive processing power offered by the large number of PEs. From the other side, the failure probability due to the technology scaling also increases. Runtime adaptation is a key to providing reliability of future SoCs. Employing only design time techniques, such as TMR, designs would have to be more and more conservative because they would spend more silicon area on the chip to provide some infrastructure to support faults. On the other hand, runtime techniques provide a graceful degradation of the system. Runtime techniques reconfigure the system to continue delivering their processing services, despite defective components due to the presence of permanent and/or transient faults throughout the system lifetime.

The Introduction of the Thesis stated the following hypothesis: "*A layered approach can cope with faults in the communication infrastructure and in the processing elements in such a way to ensure a correct operation of the MPSoC, even in the presence of multiple faults. The proposed methods increase the lifetime of current MPSoCs due to the isolation of defective routers, links, and processing elements. The remaining healthy components continue to execute the applications correctly*."

The layered approach enabled to divide the fault tolerance problem in independent parts. Each layer is responsible for solving part of the problem, as an upper layer uses the features of the lower layer to solve another part of the problem. For example, the network layer is responsible for finding a new path when a fault is detected at some router in the path. The upper layer (transport) is responsible for triggering the network layer, ensuring the correct communication between processing elements. Consequently, the first part of the above hypothesis is proven.

Results demonstrated that even with several defective routers or processing elements the MPSoC continued to execute the applications correctly, therefore, increasing the system lifetime. Such results confirm the second part of the hypothesis.

The method has original contributions in all layers, being the first proposal to addresses a comprehensive fault-tolerant approach:

- In the ***physical layer***, we propose the modification of test wrapper cells for FT purposes. With this modification, it is possible to isolate the PE and the router that is faulty, avoiding Byzantine faults. This modification can be coupled with a given test scheme to trigger the FT layered approach. In addition, this modification enabled the design of a CRC fault detection scheme in the HeMPS platform in the scope of our research group [FOC15], which is not part of the contribution of the Thesis.
- At the ***network layer***, we propose a fast path finding method, being able to find faulty-free paths in less than 40 microseconds for a worst-case scenario.
- In the ***transport layer***, different approaches were evaluated being capable of detecting a lost message and start the message retransmission. The results show that the overhead to retransmit the message is 2.46X compared to the time to transmit a message without fault, being all other messages transmitted with no overhead.
- At the ***application layer***, the entire fault recovery protocol executes fast, with a low execution time overhead with (5.67%) and with faults (17.33% - 28.34%). Table 12 resumes each layer contribution, comparing it with the OSI model.

Table 12 – Simplified view of the OSI model with the added features for fault tolerance in each layer.

Layer	Architectural Features	Fault tolerance features
Physical (Chapter 3)	Duplicated physical channels per link (16-bit flit)	CRC added to data links and CRC in routers. (in the scope of the research group [FOC15])
Data Link (Chapter 3)	Synchronous credit-based flow control	Test wrappers; packet discarding
Network (Chapter 4)	Adaptive Routing	Auxiliary NoC; degraded mode operation; faulty-free path search
Transport (Chapter 5)	Message passing (OS level); Communication API	Fault-tolerant communication protocol; packet retransmission
Application (Chapter 6)	Applications with checkpoints	Task isolating and application checkpoint/rollback

7.1 Limitations of the Current Proposal

The proposal has limitations, which are reference for future works:

- At the physical layer, the proposal requires a NoC with duplicated physical channels, in such a way to enable full adaptive routing.
- The Manager Processor (MP) is assumed fault-free. The FT communication method presented in Chapter 5 assumes the communication between PEs. Therefore, the communication between the MP and the PE is prone to faults that can lead to a system failure. The FT communication protocol should be extended to cope with faults in the path to the MP, and it should be possible to remap the MP into another processing element.
- The external memory is assumed fault-free, and the interface with this memory is a single point of failure. Replicating the external memory and the PEs with access to it could mitigate this limitation. As a second advantage, it will also remove the bottleneck to map all tasks in the system initialization.
- At the network layer, it is necessary a processor to execute the path computation. A router connected to a “passive” IP, as a memory bank, is not able to compute the deadlock-free path. A possible solution is to use the MP to handle the path computation for such IPs.
- The FT communication protocol was designed for message-passing communication. Shared memory communication requires adaptation in the proposed protocol.

7.2 Future Works

After the development of the seek network, the simulated scenarios shown the underuse of the seek network, even with multiple faults. Thus, the seek network could be used for other purposes, as a monitor for aging effects, path congestion and to transmit control packets.

The test recovery method can act as an aging monitor, being managed by the MP. If a given channel is affected by a large number of transient faults, this may signalize a wear out in the channel, and the test module may notify the MP to disable this port permanently. Note that this aging monitor is not limited to the channels of the router, but also can be used for the PEs.

A future implementation is to expand the implementation in [BAR15] to execute a Checkpoint-Recovery method for message passing. The proposal can be a mix of the approach proposed in this Thesis [WAC15] with the proposal of synchronization in [RUS08].

Another future work includes the use of more than one task repository, reducing the bottleneck to map tasks and removing the single point of failure, as discussed in the previous session.

REFERENCES

[AGA09] Agarwal, A.; Iskander, C.; Shankar, R. “Survey of Network on Chip (NoC) Architectures & Contributions”. *Journal of Engineering, Computing and Architecture*, vol. 2-1, 2009.

[ALA03] Alam, M.A. “A critical examination of the mechanics of dynamic NBTI for PMOSFETs”. In: Electron Devices Meeting, 2003, 4p.

[ALH13] Alhussien, A.; Verbeek, F.; van Gastel, B.; Bagherzadeh, N.; Schmaltz, J. “Fully Reliable Dynamic Routing Logic for a Fault-Tolerant NoC Architecture”. *Journal of Integrated Circuits and Systems*, vol. 80-1, March 2013, pp. 43-53.

[ASA06] Asanovic, K.; Bodik, R.; Catanzaro, B.; Gebis, J.; Husbands, P.; Keutzer, K.; Patterson, D.; Plishker, W.; Shalf, J.; Williams, S.; Yelick, K. “The Landscape of Parallel Computing Research: A View from Berkeley”. Technical Report, Electrical Engineering and Computer Sciences – University of California at Berkeley, 2006

[AUL04] Aulwes, R.T.; Daniel, D.J.; Desai, N.N.; Graham, R.L.; Risinger, L.D.; Taylor, Mark A.; Woodall, T.S.; Sukalski, M.W. “Architecture of LA-MPI, a network-fault-tolerant MPI”. In: Parallel and Distributed Processing Symposium, 2004, 10p.

[BAR15] Barreto, F.; Amory, A.; Moraes, F. “Fault Recovery Protocol for Distributed Memory MPSoCs”. In: ISCAS, 2015, pp. 421-424.

[BAT04] Batchu, R.; Dandass, Y. S.; Skjellum, A.; Beddhu, M. “MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware”. *Journal Cluster Computing*, vol. 7-4, October 2004, pp. 303-315.

[BEN02] Benini, L.; De Micheli, G. “Networks on chips: a new SoC paradigm”. *Computer*, vol. 35-1, January 2002, pp. 70-78.

[BEN12] Benini, L.; Flamand, E.; Fuin, D.; Melpignano, D. “P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator”. In: DATE, 2012, pp. 983–987.

[BOR07] Borkar, S. “Thousand core chips: a technology perspective”. In: DATE, 2007, pp. 746-749.

[BOY93] Boyan, J.; Littman, M. “Packet Routing in Dynamically Changing Networks: a Reinforcement Learning Approach”. In: Advances in Neural Information Processing Systems, 1993, pp. 671-678.

[BUS05] M. L. Bushnel and W. D. Agrawal, “Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits”. Boston: Springer, 2005, 690p.

[CAR08] Carara, E.; Oliveira, R. P.; Calazans, N. L. V.. “A new router architecture for High-Performance intrachip networks”. Journal Integrated Circuits and Systems, vol. 3-1, pp. 23-31, 2008.

[CAR09] Carara, E. A.; Oliveira, R. P.; Calazans, N. L V; Moraes, F. G. “HeMPS - a framework for NoC-based MPSoC generation”. In: In: ISCAS, 2009, p. 1345-1348.

[CAR10] Carara, E.; Moraes, F. “Flow Oriented Routing for NOCS”. In: SOCC, 2010, pp. 367-370.

[CAS14] Castilhos, G.; Wachter, E.W.; Madalozzo, G.A.; Erichsen, A.; Monteiro, T.; Moraes, F.G. “A Framework for MPSoC Generation and Distributed Applications Evaluation”. In: ISQED, 2014. 4p.

[CHI11] Wang, C.; Hu, W.; Lee, S.; Bagherzadeh, N. “Area and power-efficient innovative congestion-aware Network-on-Chip architecture”. *Journal of Systems Architecture*, vol. 57-1, January 2011, pp. 24-38.

[CLE13] CLEARSPEED. “CSX700”. Available at: http://www.clearspeed.com/products/csx700.php. December 2013.

[CON09] Concatto, C.; Matos, D.; Carro, L.; Kastensmidt, F.; Susin, A.; Cota, E.; Kreutz, M. “Fault tolerant mechanism to improve yield in NoCs using a reconfigurable router”. In: SBCCI, 2009, 6p.

[COP08] Coppola, M.; Grammatikakis, M.; Locatelli R.; Maruccia, G.; Pieralisi L. “Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC”. CRC Press, 2008, 288 p.

[COT12] Cota, É.; Amory, A. M.; Lubaszewski, M. S. “Reliability, Availability and Serviceability of Networks-on-Chip”. Springer, 2012, 209p.

[DEO12] DeOrio, A.; Fick, D.; Bertacco, V.; Sylvester, D; Blaauw, D.; Jin Hu; Chen, G. “A Reliable Routing Architecture and Algorithm for NoCs”. *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 31-5, 2012, pp. 726-739.

[DOO14] Doowon Lee; Parikh, R.; Bertacco, V. “Brisk and limited-impact NoC routing reconfiguration”. In: DATE, 2014, pp. 1-6.

[EBR13] Ebrahimi, M.; Daneshtalab, M.; Plosila, J.; Tenhunen, H., “Minimal-path fault-tolerant approach using connection-retaining structure in Networks-on-Chip”. In: NOCS, 2013, 4p.

[FEE13] Feehrer, J.; Jairath, S.; Loewenstein, P.; Sivaramakrishnan, R.; Smentek, D.; Turullols, S.; Vahidsafa, A. “The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets”. *IEEE Micro*, vol. 33-2, Mar-Apr 2013, pp. 48-57.

[FEN10] Feng, C.; Lu, Z.; Jantsch, A.; Li, J.; Zhang, M. “A Reconfigurable Fault-Tolerant Deflection Routing Algorithm Based on Reinforcement Learning for Network-on-Chip”. In: NoCArc, 2010, pp. 11-16.

[FIC09] Fick, D.; DeOrio, A.; Chen, G.; Bertacco, V.; Sylvester, D; Blaauw, D. “A Highly Resilient Routing Algorithm for Fault-Tolerant NoCs”. In: DATE, 2009, pp. 21-26.

[FLI07] Flich, J.; Mejia, A.; Lopez, P.; Duato, J. “Region-Based Routing: An Efficient Routing Mechanism to Tackle Unreliable Hardware in Network on Chips”. In: NOCS, 2007, pp. 183-194.

[FLI12] Flich, J.; Skeie, T.; Mejia, A.; Lysne, O.; Lopez, P.; Robles, A.; Duato, J.; Koibuchi, M.; Rokicki, T.; Sancho, J.C. “A Survey and Evaluation of Topology-Agnostic Deterministic Routing Algorithms”. *IEEE Transactions on Parallel and Distributed Systems*, vol. 23-3, March 2012, pp.405-425.

[FOC15] Fochi, V.; Wachter, E.; Erichsen, A.; Amory, A.; Moraes, F. “An Integrated Method for Implementing Online Fault Detection in NoC-Based MPSoCs”. In: ISCAS, 2015, 1562-1565.

[FU10] Fu, F.; Sun, S.; Hu, X.; Song, J.; Wang, J.; Yu, M. “MMPI: A flexible and efficient multiprocessor message passing interface for NoC-based MPSoC”. In: SOCC, 2010, pp. 359-362.

[GAR13] Garibotti, R.; Ost, L.; Busseuil, R.; Kourouma, M.; Adeniyi-Jones, C.; Sassatelli, G.; Robert, M. “Simultaneous multithreading support in embedded distributed memory MPSoCs”. In: DAC, 2013, 7p.

[GE00] Ge-Ming C. “The odd-even turn model for adaptive routing”. *IEEE Transactions on Parallel and Distributed Systems*, vol.11-7, July 2000, pp. 729-738.

[GHA82] Ghate, P.B. “Electromigration-Induced Failures in VLSI Interconnects”. In: Reliability Physics Symposium, 1982, 8p.

[GIZ11] Gizopoulos, D.; Psarakis, M.; Adve, S.V.; Ramachandran, P.; Hari, S.K.S.; Sorin, D.; Meixner, A.; Biswas, A.; Vera, X. “Architectures for online error detection and recovery in multicore processors”. In: DATE, 2011, 6p.

[GON08] Gong, R.; Dai, K.; Wang, Z. “Transient Fault Recovery on Chip Multiprocessor based on Dual Core Redundancy and Context Saving”. In: ICYCS, 2008, pp. 148–153.

[HEB11] Hebert, N.; Almeida G. M.; Benoit, P.; Sassatelli G.; Torres, L. “Evaluation of a Distributed Fault Handler Method for MPSoC”. In: ISCAS, 2011, pp. 2329-2332.

[HEN13] Henkel, J.; Bauer, L.; Dutt, N.; Gupta, P.; Nassif, S.; Shafique, M.; Tahoori, M.; Wehn, N. “Reliable on-chip systems in the nano-era: Lessons learnt and future trends”. In: DAC, 2013, 10p.

[IBM13] IBM. “The Cell Project”. Available at: https://www.research.ibm.com/cell/cell_chip.html. December 2013

[INT13a] INTEL. “Teraflops Research Chips”. Available at: http://www.intel.com/pressroom/kits/Teraflops/index.htm. December 2013.

[INT13b] INTEL, “Single-Chip Cloud Computer: Project”. Available at: http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html. December 2013

[ITR15] International Technology Roadmap for Semiconductors. “ITRS 2011 edition”. Accessed in: http://www.itrs.net/reports.html. April 2015.

[JAV14] Javaid, H.; Parameswaran, S. “Pipelined Multiprocessor System-on-Chip for Multimedia”. Springer, 169p. 2014.

[JER05] Jerraya, A. A.; Wolf, W. “Multiprocessor Systems-on-Chips”. Morgan Kaufmann, 2005, 602p.

[KAR09] Kariniemi, H.; Nurmi, J. “Fault-Tolerant Communication over Micronmesh NOC with Micron Message-Passing Protocol”. In: SOC, 2009, pp. 5–12.

[KER14] Kerkhoff, H.G.; et al. “Linking aging measurements of health-monitors and specifications for multi-processor SoCs”. In: DTIS, 2014. pp. 1-6.

[KOI08] Koibuchi, M.; Matsutani, H.; Amano, H.; Pinkston, T. M. “A Lightweight Fault-Tolerant Mechanism for Network-on-Chip”. In: NOCS, 2008, 10p.

[LEE61] Lee, C. Y. “An Algorithm for Path Connections and Its Applications”. *Transactions on Electronic Computers*, September 1961, pp. 346-365.

[LHU14] Lhuillier, Y.; Ojail, M.; Guerre, A.; Philippe, J.; Chehida, K.; Thabet, F.; Andriamisaina, C.; Jaber, C.; David, R. “HARS: A hardware-assisted runtime software for embedded many-core architectures”. ACM Transactions on Embedded Computing, 2014, 25 p.

[LI12] Li, T.; Ragel, R.; Parameswaran, S. “Reli: hardware/software checkpoint and recovery scheme for embedded processors”. In: DATE, 2012, pp 875–880.

[LIN91] Linder, D.H.; Harden, J.C. “An Adaptive and Fault Tolerant Wormhole Routing Strategy for k-ary n-cubes”. *Transactions on Computers*, vol. 40-1, 1991, pp. 2-12.

[LUC09] Lucas, A.; Moraes, F. “Crosstalk fault tolerant NoC - design and evaluation”. In: IFIP VLSI-SOC, 2009, 6 p.

[MAH08] Mahr, P.; Lorchner, C.; Ishebabi, H.; Bobda, C. “SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips”. In: International Conference on Reconfigurable Computing and FPGAs, 2008, pp. 187-192.

[MAR09] Marinissen, E.J.; Zorian, Y., “IEEE Std 1500 Enables Modular SoC Testing”. *Design & Test of Computers,* vol. 26-1, Jan-Feb 2009, pp. 8-17.

[MAR13] Maricau, E.; Gielen, G. “Analog IC Reliability in Nanometer CMOS”. Springer, 2013.

[MAT13] Matos, D.; Concatto, C.; Kologeski, A.; Carro, L.; Kastensmidt, F.; Susin, A.; Kreutz, M. “A NOC closed-loop performance monitor and adapter”. *Journal Microprocessors & Microsystems*, vol. 37-6, August 2013, pp. 661-671.

[MCL00] McLachlan, G. J.; Peel, D. “Finite mixture models”. New York: Wiley, 2000.

[MEL12] Melpignano, D.; Benini, L.; Flamand, E.; Jego, B.; Lepley, T.; Haugou, G.; Clermidy, F.; Dutoit, D. “Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications”. In: Design Automation Conference (DAC), 2012, pp.1137-1142.

[NVI13] NVIDIA, “NVIDIA KEPLER GK110 Next-Generation CUDA Compute Architecture”. Available at: http://www.nvidia.com/object/nvidia-kepler.html. December 2013.

[PET12] Petry, C.; Wachter, E.W.; Castilhos, G.; Moraes, F.G.; Calazans, N. “A Spectrum of MPSoC Models for Design Space Exploration and Its Use”. In: RSP, 2012, pp. 30-35.

[PRV02] Prvulovic, M.; Zheng Zhang; Torrellas, J. “ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors”. In: ISCA, 2002, pp. 111–122.

[PSA10] Psarakis, M.; et al. “Microprocessors software-based self-testing,” IEEE Design and Test of Computers, v. 27(3), 2010, pp. 4-19.

[RAD13] Radetzki, M.; Feng, C.; Zhao, X.; Jantsch, A. “Methods for Fault Tolerance in Networks on Chip”. *ACM Computing Surveys*, vol. 46-1, October 2013, 38p.

[ROD09a] Rodrigo, S.; Medardoni, S.; Flich, J.; Bertozzi, D.; Duato, J. “Efficient implementation of distributed routing algorithms for NoCs”. *Computers & Digital Techniques*, vol.3-5, September 2009, pp.460-475.

[ROD09b] Rodrigo, S.; Hernandez, C.; Flich, J.; Silla, F.; Duato, J.; Medardoni, S.; Bertozzi, A D.; Mej, A. Dai, D. “Yield-oriented evaluation methodology of network-on-chip routing implementations”. In: SoC, 2009, pp. 100-105.

[ROD11] Rodrigo, S.; Flich, J.; Roca, A.; Medardoni, S.; Bertozzi, D.; Camacho, J.; Silla, F.; Duato, J. “Cost-Efficient On-Chip Routing Implementations for CMP and MPSoC Systems”. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30-4, 2011, pp. 534-547.

[RUS08] Rusu, C.; Grecu, C.; Anghel, L. “Improving the scalability of checkpoint recovery for networks-on-chip”. In: ISCAS 2008. pp.2793-2796.

[SAL08] Salminen, E.; Kulmala, A. ; Hamalainen, T. D. “Survey of Network-on-Chip Proposals”. White Paper OCP-IP, 2008, 13p.

[SAN10] Nassif, S. R.; Mehta, N.; Cao, Y. “A resilience roadmap”. In: DATE, 2010, 6p.

[SCH07] Schonwald, T.; Zimmermann, J.; Bringmann, O.; Rosenstiel, W. “Fully Adaptive Fault-Tolerant Routing Algorithm for Network-on-Chip Architectures”. In: Euromicro, 2007, pp. 527-534.

[SEM11] Sem-Jacobsen, F.; Rodrigo, S.; Skeie, T. “iFDOR: Dynamic Rerouting on-Chip”. In: International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip, 2011, pp 11-14.

[SHA14] Shafik, R. A.; Al-Hashimi, B. M.; Chakrabarty, K. “System-Level Design Methodology. In Energy-Efficient Fault-Tolerant Systems”. New York: Springer, 2014, pp. 169-210.

[SKE09] Skeie, T.; Sem-Jacobsen, F.O.; Rodrigo, S.; Flich, J.; Bertozzi, D.; Medardoni, S. “Flexible DOR routing for virtualization of multicore chips”. In: SOC, 2009, pp. 73-76.

[SOU13] Sourdis, I.; Strydis, C.; Armato, A.; Bouganis, C.; Falsafi, B.; Gaydadjiev, G.; Isaza, S.; Malek, A.; Mariani, R.; Pnevmatikatos, D.; Pradhan, D.K.; Rauwerda, G.; Seepers, R.; Shafik, K.; Sunesen, K.; Theodoropoulos, D.; Tzilis, S.; Vavouras, M. “DeSyRe: On-demand system reliability”. Microprocessors and Microsystems. 37, 8, 2013, pp 981–1001.

[STA11] Stanisavljević, M.; Schmid, A.; Leblebici, Y. “Reliability of Nanoscale Circuits and Systems - Methodologies and Circuit Architectures”. Springer, 2011, 195p.

[SUT13] Sutter, H. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”. Available at: http://www.gotw.ca/publications/concurrency-ddj.htm. December 2013.

[TAN06] Tanenbaum, A. S.; Woodhull, A. S. “Operating systems: design and implementation”. Pearson Prentice Hall, 2006. 1054 p.

[THO10] Thonnart, Y.; Vivet, P.; Clermidy, F. “A fully-asynchronous low-power framework for GALS NoC integration”. In: DATE 2010.

[TIL13] TILERA. “TILE-Gx Processor Family”. Available at: http://www.tilera.com/products/processors/TILE-Gx_Family. December 2013.

[WAC11a] Wachter, E. W. “Integração de Novos Processadores em Arquiteturas MPSOC: um Estudo de Caso”. Master Dissertation, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2011, 92p.

[WAC11b] Wachter, E. W.; Biazi, A.; Moraes, F.G. “HeMPS-S: A Homogeneous NoC-Based MPSoCs Framework Prototyped in FPGAs”. In: ReCoSoC, 2011, 8p.

[WAC12a] Wachter, E. W.; Carlo, L.; Carara, E.; Moraes, F.G. “An Open-source Framework for Heterogeneous MPSoC Generation”. In: SPL, 2012, 4p.

[WAC12b] Wachter, E. W.; Moraes, F.G. “MAZENOC: Novel Approach for Fault-Tolerant NoC Routing”. In: SOCC, 2012, 6p.

[WAC13a] Wachter, E.W.; Erichsen, A.; Amory, A.M.; Moraes, F.G. “Topology-Agnostic Fault-Tolerant NoC Routing Method”. In: DATE, 2013, 6p.

[WAC13b] Wachter, E.W.; Amory, A.M.; Moraes, F.G. Fault Recovery Communication Protocol for NoC-based MPSoCs In: ISVLSI PhD Forum, 2013, 2p.

[WAC14a] Wachter, E.W.; Erichsen, A.; Juracy, L.; Amory, A.M.; Moraes, F.G. “Runtime Fault Recovery Protocol for NoC-based MPSoCs”. In: ISQED, 2014, 8p.

[WAC14b] Wachter, E.W.; Erichsen, A.; Juracy, L.; Amory, A.M.; Moraes, F.G. “A Fast Runtime Fault Recovery Approach for NOC-Based MPSOCS for Performance Constrained Applications”. In: SBCCI, 2014, 7p.

[WAC15] Wachter, E.; Ventroux, N.; Moraes, F. “A Context Saving Fault Tolerant Approach for a Shared Memory Many-Core Architecture”. In: ISCAS (International Symposium on Circuits and Systems), 2015. 4p

[WAN06] Wang, L-T; Wu, C-W; Wen, X. “VLSI test principles and architectures: design for testability”. Academic Press, 2006.

[WOS07] Woszezenki, C. “Alocação de Tarefas e Comunicação entre Tarefas em MPSoCs”. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2007, 121p.

[ZHU06] Zhu, X.; Qin, W. “Prototyping a Fault-Tolerant Multiprocessor SoC with Run-Time Fault Recovery”. In: DAC, 2006, pp. 53–56.

APPENDIX 1 – LIST OF PUBLICATIONS

Table 13 presents the set of publications held since the beginning of the PhD. The description column links the paper to this work section, when applicable, or to the main theme of the publication.

Table 13 – Publications during the PhD period.

	Publication	Description
1	***HeMPS-S: A Homogeneous NoC-Based MPSoCs Framework Prototyped in FPGAs*** Wächter, E.W.; Biazi, A.; Moraes, F.G. In: ReCoSoC, 2011.	Prototyping of the reference architecture
2	***Exploring Heterogeneous NoC-based MPSoCs: from FPGA to High-Level Modeling*** Ost, L.; Almeida, G.M.; Mandelli, M.; Wächter, E.W.; Varyani, S.; Indrusiak, L.; Sassatelli, G.; Robert, M.; Moraes, F.G. In: ReCoSoC, 2011.	Analysis of different applications in different organizations of heterogeneous MPSoCs
3	***Exploring Adaptive Techniques in Heterogeneous MPSoCs based on Virtualization*** Ost, L.; Varyani, S.; Mandelli, M.; Almeida, G.; Indrusiak, L.; Wächter, E.W.; Moraes, F.G.; Sassatelli, G. ACM Trans. on Reconfigurable Technology and Systems, v. 5, p. 17:1-17:11, 2012.	Analysis of different applications in different organizations of heterogeneous MPSoCs
4	***An Open-source Framework for Heterogeneous MPSoC Generation*** Wächter, E.W.; Carlo, L.; Carara, E.; Moraes, F.G. In: SPL, 2012	Framework of the reference architecture
5	***A Spectrum of MPSoC Models for Design Space Exploration and Its Use*** Petry, C.; Wächter, E.W.; Castilhos, G.; Moraes, F.G.; Calazans, N. In: RSP, 2012	Analysis of the different models of the reference MPSoC
6	***MAZENOC: Novel Approach for Fault-Tolerant NoC Routing*** Wächter, E.W.; Moraes, F.G. In: SOCC, 2012	Chapter 4
7	***Topology-Agnostic Fault-Tolerant NoC Routing Method*** Wächter, E.W.; Erichsen, A.; Amory, A.M.; Moraes, F.G. In: DATE, 2013	Chapter 4
8	***Fault Recovery Communication Protocol for NoC-based MPSoCs*** Wächter, E.W.; Amory, A.M.; Moraes, F.G. In: ISVLSI, 2013, PhD Forum, 2013.	First Method of Chapter 5
9	***Runtime Fault Recovery Protocol for NoC-based MPSoCs*** Wächter, E.W.; Erichsen, A.; Juracy, L.; Amory, A.M.; Moraes, F.G. In: ISQED, 2014.	First Method of Chapter 5
10	***A Framework for MPSoC Generation and Distributed Applications Evaluation*** Castilhos, G.; Wächter, E.W.; Madalozzo, G.A.; Erichsen, A.; Monteiro, T.; Moraes, F.G. In: ISQED, 2014.	Toolset to generate the reference architecture
11	***A Fast Runtime Fault Recovery Approach for NOC-Based MPSOCS for Performance Constrained Applications*** Wächter, E. W.; Erichsen, A.; Juracy, L.; Amory, A. M; Moraes, F. G. In: SBCCI, 2014.	Second Method of Chapter 5
12	***A Context Saving Fault Tolerant Approach for a Shared Memory Many-Core Architecture*** Wachter, E.; Ventroux, N.; Moraes, F. In: ISCAS, 2015. 4p.	Approach developed in the sandwich thesis – Chapter 6
13	***An Integrated Method for Implementing Online Fault Detection in NoC-Based MPSoCs*** Fochi, F.; Wächter, E.; Erichsen, A.; Amory, A.; Moraes, F. In: ISCAS, 2015. 4p.	Example of possible fault detection to explore in Chapter 3