PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIAS DA COMPUTAÇÃO

VINICIUS MORAIS FOCHI

FAULT-TOLERANCE AT THE MANAGEMENT LEVEL IN MANY-CORE SYSTEMS

Porto Alegre
2019

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica do Rio Grande do Sul

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

FAULT-TOLERANCE AT THE MANAGEMENT LEVEL IN MANY-CORE SYSTEMS

VINICIUS M. FOCHI

Thesis presented as partial requirement for obtaining the degree of PhD in Computer Science at Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: Prof. Fernando Gehm Moraes

Porto Alegre
2019

Ficha Catalográfica

F652f Fochi, Vinicius Morais

Fault-Tolerance at the Management Level in Many-Core Systems / Vinicius Morais Fochi . – 2019.
108 p.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. MCSoCs. 2. System Management. 3. Fault-Tolerance. 4. Application Admission. 5. BrNoC. I. Moraes, Fernando Gehm. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Vinicius Morais Fochi

Fault-Tolerance at the Management Level in Many-Core Systems

Tese apresentada como requisito parcial para obtenção do grau de Doutor em Ciências da Computação do Programa de Pós-Graduação em Ciências da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado em 23 de agosto de 2019.

BANCA EXAMINADORA:

Prof. Dra. Fernanda Kastensmidt – Avaliadora (UFRGS)

Prof. Dr. Eduardo Bezerra – Avaliador (UFSC)

Prof. Dr. Cesar Marcon – Avaliador (PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Orientador)

ACKNOWLEDGMENTS

Gostaria de agradecer as pessoas que tiveram influência sobre o desenvolvimento desta tese e de todo meu Doutorado. Gostaria de dedicar esta conquista aos meus pais, José Jorge Fochi e Rosi Morais Fochi que me apoiaram a continuar estudando e a minha irmã Cristina.

Ao Professor Fernando Gehm Moraes, por aceitar o imenso desafio de continuar me orientando. Muito obrigado por manter reuniões semanais, me obrigando assim a trabalhar intensamente no desenvolvimento do trabalho. Obrigado por todo o conhecimento compartilhado e por sempre acreditar no nosso trabalho. Por ter compartilhado seu tempo e paciência para a realização desta tese. Sou muito grato por me aceitar como aluno de doutorado.

Agradeço a todos os professores do PPGCC pelo conhecimento compartilhado. Aos colaboradores do PPGCC pela dedicação em nos manter sempre informados dos prazos e burocracias.

Um agradecimento especial ao colega Luciano Lores Caimi, grande colaborador no desenvolvimento deste trabalho e organizar todos os eventos.

Aos colegas que me ajudaram durante a jornada do Doutorado: Eduardo Wachter, Marcelo Ruaro, L. Heck, G. Heck, Roger, Juraci, Marcelo Holgado e a todos os demais: obrigado pela ajuda!

Agradeço a Capes e os contribuintes por terem possibilitado e financiado esta pesquisa.

TOLERÂNCIA A FALHAS NO NÍVEL DA GERERÊNCIA EM SISTEMAS MANY-CORE

RESUMO

A redução dos nodos tecnológicos permitiu o surgimento de sistemas com múltiplos núcleos de processamento utilizando redes intra-chip (MCSoCs - many-core systems-on-chip), com dezenas a centenas de elementos de processamento (PEs). Apesar do poder de processamento oferecido pelo grande numero de PEs e da flexibilidade de comunicação devido à adoção de NoCs, é necessário gerenciar os recursos do sistema para garantir sua escalabilidade. A execução das tarefas de gerência requer PEs reservados exclusivamente para executar essas ações. Uma abordagem centralizada induziria uma carga de trabalho significativa para os PEs de gerência (MPE) em sistemas de grande escala. A adoção de abordagens distribuídas, com MPEs hierarquicamente organizadas, reduz a carga de gerência, sendo a organização adotada nesta Tese. Propostas recentes de gerência em MCSoCs focam em diferentes aspectos: potência, desempenho, utilização dos recursos do sistema. Essas técnicas são aplicadas no nível sistêmico dos MCSoCs. No entanto, nos trabalhos analisados, há uma lacuna nas propostas relacionadas a falhas permanentes nos MPEs. Esta Tese tem por objetivo abordar dois problemas principais. Primeiro, tratar falhas permanentes nos MPEs, desenvolvendo um conjunto de novas técnicas para que os MCSoCs continuem a operar corretamente, sem reexecutar as aplicações em execução. Segundo, resolver a questão do ponto único de falha na comunicação dos MCSoCs com o mundo externo. A contribuição original desta Tese é uma arquitetura MCSoC distribuída, com capacidade de recuperação de falhas em pontos críticos do sistema. O método de recuperação inclui módulos de hardware e software, monitoramento de falhas e recuperação de gerenciamento. A proposta utiliza técnicas de migração de tarefas e heurísticas para selecionar a posição do novo MPE. Esta Tese propõe um método de recuperação quando um MPE torna-se falho. O método é escalável, capaz de atuar em sistemas de dezenas a centenas de processadores. O método é transparente para as aplicações executadas no MCSoC, com uma pequena sobrecarga no tempo de execução, observado durante a migração de gerência e migração de tarefas.

Palavras-Chave: MCSoCs; NoC; Gerenciamento de sistemas; Recuperação de falhas; Migração de tarefas; Tolerância a falhas; Admissão de aplicativos; BrNoC.

FAULT-TOLERANCE AT THE MANAGEMENT LEVEL IN MANY-CORE SYSTEMS

ABSTRACT

The technology nodes reduction enabled the emergence of NoC-based many-cores with dozens to hundreds of processing elements (PEs). Despite the processing power offered by a large number of processors and communication flexibility due to the adoption of NoCs, it is necessary to manage the many-core resources to ensure scalability. The execution of the management tasks requires processing elements reserved exclusively to execute such actions. A centralized approach would induce a significant load to the managers PEs (MPE) in large-scale systems. The adoption of distributed approaches, with MPEs hierarchically organized, reduces the management load, being the organization adopted in this work. Recent proposals for Many-core System-on-chip (MCSoCs) management focus on different aspects: power, performance, system resources. These management techniques are applied to the systemic level of the MCSoCs. However, in the reviewed works, there is a gap in proposals related to permanent faults in processors with management functions. This Thesis aims to tackle two main problems. First, to treat permanent faults in management processors, developing a set of new techniques so that the MCSoCs continues to operate correctly, without re-executing applications running on it. Second, to solve the single point of failure issue regarding the communication of the MCSoCs with the external world. The original contribution of this Thesis is a distributed MCSoC architecture, with fault recovery capability at critical points in the system. The recovery method includes hardware and software modules, fault monitoring, and management recovering. The proposal uses task migration techniques, and heuristics to select the position of the new manager. This Thesis proposes a recovery method when an MPE became faulty. The method is scalable, able to act in systems from dozens up to hundreds of processors. The method is transparent to the applications executing in the MCSoC, with a small execution overhead observed during the management and task migration.

Keywords: MCSoCs; NoC; System Management; Fault-recovery; Task migration; Fault-tolerance; Application Admission, BrNoC.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACT – Autonomous Chip Tester

AET – Application Execution Time

AIM – Application Injector Machine

API – Application Program Interface

BRT – Broker Recovery Time

BSS – Block Started by Symbol

CAM – Content Addressable Memory

CRC – Cyclic Redundancy Check

DMA – Direct Memory Access

DMNI – Direct Memory Network Interface

DVFS – Dynamic Voltage and Frequency Scaling

DWC – Duplication with comparison

DWCR – Duplication with comparison and re-execution

ECC – Error Correction Codes

EBL – Effective Buffer Length

FIFO – First In First Out

FPD – fault-resilient Packet delivery

FPGA – Field Programmable Gate Array

FTR – Fault-Tolerant Routing

FSM – Finite State Machine

FWM – Fail Wrapper Module

GMP – Global Manager Processor

GM – Global Manager

GPPC – General Purpose Processing Cores

GPD – General Purpose Device

HEMPS – Hermes MultiProcessor System

I/O – Input/Output

IP – Intellectual Property

LMP – Local Manager Processor

LM – Local Manager

MEMPHIS – Many-core Modeling Platform for Heterogenous SoCs

MP – Manager Processor

MPSOC – Multi-Processor System-on-Chip

MCSOC – Many-Core System-on-Chip

NI – Network Interface

NOC – Network On Chip

NBTI – Negative Bias Temperature Instability

ODA – Observe-decide-act

OS – Operating System

PE – Processor Element

PDA – Path-Diversity-Aware

PPN – Polyhedral Process Network

PS – Packet Switch

QOS – Quality Of service

RISC – Reduced Instruction Set Computing

RTL – Register Transfer Level

REM – Runtime Energy Management

RTOS – Real-Time Operating System

SCC – Single-chip Cloud Computer

SA0 – Stuck-At-0

SA1 – Stuck-At-1

SR – Source Routing

SP – Slave Processor

SOC – System-On-Chip

SBST – Software-Based Self-Test

SAF – Stuck-at Faults

TCB – Task Control Block

TMR – Triple Modular Redundancy

TRA – Test Response Analyzer

TMSU – Task Mapping and Scheduling Unit

TM – Task Mapper

TS – Task Scheduler

VHDL – VHSIC Hardware Description Language

VLSI – Very Large Scale Integration

VHSIC – Very High Speed Integrated Circuit

WET – Workload Execution Time

CONTENTS

1. INTRODUCTION

The continuous development in Very Large Scale Integration (*VLSI*) technology and scaling of transistors to nanometer range led to the integration of billions of transistors on a single chip. This allows the system designer to embed a large number of intellectual property cores, memory units, and Processing Elements (*PE*s) onto a single chip resulting in a System-on-Chip (*SoC*).

Shared buses are commonly used for the communication between different computation and memory units present in the *SoC*. However, the performance of on-chip communication medium between the components becomes a critical issue with the increase in the number of cores [Grecu et al., 2004]. Networks-on-chip (*NoC*s) have been proposed as a viable solution to deal with this limitation [Benini and Micheli, 2002]. In the *NoC* paradigm, data communication among various cores is achieved through an on-chip network consisting of routers and links. In a direct *NoC* topology, each core is attached to a router through a Network Interface (*NI*) module. *NoC*s provides a scalable, flexible, and reusable communication infrastructure, which is required for *SoC*s where different on-chip elements need to communicate with each other in parallel.

Large *SoC*s require processing elements (*PE*s) dedicated to management purposes, for example, execute the task mapping, handle monitoring data obtained from sensors and estimation functions, and run self-awareness adaptation (e.g., quality-of-service, *DVFS* control, aging, temperature) [Dutt et al., 2015, Tajik et al., 2016]. Many-core Systems-on-Chip (*MCSoCs*) with a hierarchical organization ensure scalability at the management level, with *PE*s having distinct roles: managers (*MP*), responsible for manipulating system resources at runtime, and slave processors (*SP*), processors that only executed tasks, [Faruque et al., 2008]. With such organization, *MCSoCs* contains virtual regions, named clusters, with one *MP* and a set of *SP*s per cluster. A cluster may increase its size at runtime, borrowing *SP*s from neighbor clusters, in a process named re-clustering [Castilhos et al., 2013].

Transistors, vias, and wires degrade faster over time in deep sub-micron technologies, inducing transient faults and permanent faults, thus shortening integrated circuits lifetime [Kim et al., 2013]. The aggressive technology scaling and increasing design complexity of *MCSoCs* made the chip components vulnerable to faults. The failure rate of electronic components increases as high as 316% with 64% decrease in the feature size [Srinivasan et al., 2004]. These faults may be permanent or transient. Permanent faults are nonrecoverable device defects. These faults can occur due to manufacturing defects or device wear-out caused by Negative Bias Temperature Instability (*NBTI*) [Knebel et al., 2016], electromigration and oxide breakdown [Fick et al., 2009a].

While, transient faults are temporary random faults that may occur for a short interval of time due to, e.g., crosstalk [Cui et al., 2016], alpha particles, cosmic radiation [Li and Draper, 2016], permanent faults damage the on-chip elements. As a result, components with permanent faults are no longer available to the system, inducing performance degradation, system malfunction, or compromising the entire chip.

Thus, reliability becomes a key issue in *MCSoC* design [Heron et al., 2010]. Classical fault-tolerant approaches, as Triple Modular Redundancy (*TMR*) or spare components [Reddy et al., 2016], do not comply with today's requirements of silicon area and power dissipation. Due to the way it is built, an *MCSoC* provides a set of replicated structures (*PE*s), where a healthy component can execute the faulty component functions, resulting in graceful performance degradation.

It is worth to differentiate the consequences of a permanent fault in *SP*s and in *MP*s. A fault in a PE executing a user application (*SP*) compromises the application, being possible to remap the application [Barreto et al., 2015]. The effect of a fault in an *MP* is more severe than a fault in an SP, because it may halt the entire cluster, making the set of *SP*s controlled by the faulty MP unavailable.

1.1 Motivation

Modern *MCSoCs* increasingly require runtime fault recovery methods because fault probability increases in deep-sub-micron technology nodes. Thus, systems should adopt self-adaptation techniques to cope with transient and permanent faults to extend the system lifetime.

System management, and how to deal with a failure on it, opens a set of new challenges and opportunities in the field of many-core systems research. Several system-level approaches are available in the literature: as power management (*DVFS*)[Martins et al., 2016], performance/quality-of-service (*QoS*) management [Ruaro and Moraes, 2017], resource management[Paul et al., 2015]. A rich literature with methods to test the processing elements modules is also available, with approaches adopted at different levels (hardware or software) or modules (as *NoC*[Fochi et al., 2015], processors, memories [Meloni et al., 2012]).

However, there is a gap in the literature related to fault-tolerant methods at the system level, i.e., related to the processors with the function to manage the system. Therefore, system management requires alternative monitoring and actuation policies to recover the system when one of these processors presents a permanent fault.

Besides fault-tolerant methods at the system level, the deployment of new applications into the *MCSoCs* is a subject with few works in the literature. The availability of distributed *MCSoCs* architectures adds flexibility to define new methods to deploy applications into the system. For example, it becomes possible to consider several instances responsible for injecting new applications into the *MCSoC*, eliminating the single point of failure when only one device is in charge to deploy applications.

In the context of this Thesis, the focus of the research is the investigation of techniques to deal with permanent faults in *MP* and methods to deploy applications into the *MCSoC*. Fault detection is out of the scope of the Thesis.

1.2 Problem Definition

This Session initially presents the *MCSoC* reference architecture with the goal to identify to the reader the relevant features of modern systems. Next, the text presents the challenges related to fault-tolerance covered in this Thesis.

1.2.1 Reference Architecture

This Section presents the reference architecture. The Thesis adopts it for the proof-of-concept of proposed methods. It is worth to mention the methods proposed throughout this Thesis are not specific to this architecture, but generic, with applicability in systems with similar architectural features.

The reference many-core platform (Figure 1.1) has the following features [Carara et al., 2009, Castilhos et al., 2013, Ruaro et al., 2019]:

- NoC-based system: the Network-on-Chip (*NoC*) connection allows multiple communications between *PE*s while ensures scalability. The *NoC* adopts 2D-mesh topology, input buffering, credit-based flow control, round-robin arbitration, wormhole packet-switching, support for deterministic XY and source routing, 8-flit buffer depth, input buffering, and duplicated physical channels (two 16-bit channels per link), enabling full adaptive routing.
- Homogeneous *GPPC* (General Purpose Processing Core): all *PE*s have the same hardware architecture with a router, a private memory, an MIPS-like processor and a Direct Memory Network Interface (*DMNI*) module.
- Distributed memory: each *PE* has a private memory, responsible for storing instructions and data. Inter-task communication occurs through message-passing.
- Applications modeled as task graphs: the applications are divided into tasks, and a graph defines the communication flow between them.

- Distributed management: the system is divided into clusters. Every cluster contains a Local Manager *PE* (*LM*), which manages the cluster, and a set of Slave *PE*s (*SP*s), which runs the applications' tasks. One of the *LM*s has the role of Global Manager (*GM*), being the only PE with access to external devices (e.g. application repository). The *GM* works as a *LM* and distributes the applications to clusters. The Operating System (*OS*) running on *PE*s defines their role in the system.

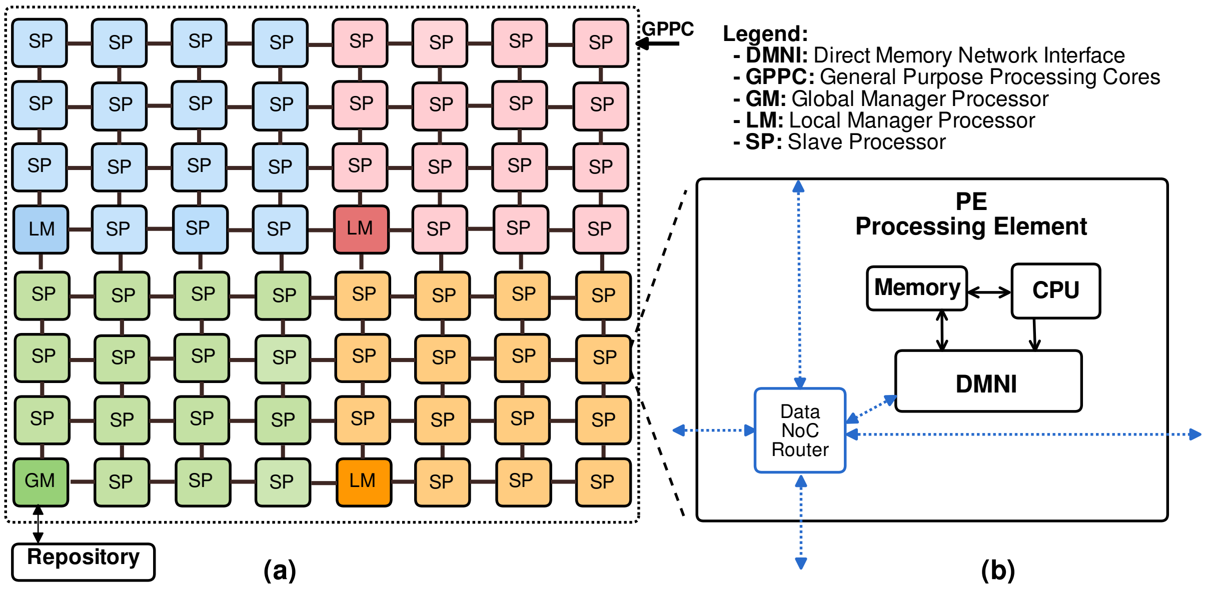

SP
LM
GM
GPPC
Repository
(a)
Legend:
- DMNI: Direct Memory Network Interface
- GPPC: General Purpose Processing Cores
- GM: Global Manager Processor
- LM: Local Manager Processor
- SP: Slave Processor
PE
Processing Element
Memory
CPU
DMNI
Data NoC Router
(b)

Figure 1.1 – Overview of *MCSoC* hardware model adopted as reference for this Thesis.

1.2.2 Problem Definition

Recent proposals for many-core management focus on different aspects: power [Martins et al., 2016, Haghbayan et al., 2014], performance [Bolchini et al., 2013], system resources [Paul et al., 2015]. These management techniques are applied to the systemic level of the *MCSoC*. However, in the reviewed works, there is a gap in proposals related to permanent faults in processors with management functions.

For fault-tolerance, the literature present proposals at different levels: processor level [Braak et al., 2010, Walters et al., 2011], router level [Yu et al., 2011, Chen et al., 2017], link level [Vitkovskiy et al., 2012, Veiga and Zeferino, 2010]. It is important to differentiate the impact of a permanent fault in a *SP* and in a manager processor (*GM* and *LM*). A faulty *SP* may be isolated using wrappers and the tasks assigned to it remapped to another SP. A faulty manager processor compromises all *SP*s of the cluster by compromising the computational capacity of the system, but the system may continue to operate. The existence of a *LM* with a global management function (*GM*) represents a single point of failure.

This Thesis aims to tackle two main problems. First, to treat **permanent** faults in management processors (*LM* and *GM*), developing a set of new techniques so that the *MCSoC* continues to operate correctly, without re-executing applications running on it. Second, to solve the single point of failure issue regarding the communication of the *MCSoC* with the external world.

1.2.3 Thesis Statement

Hardware and software faults in *MCSoCs* may compromise not only the execution of the applications, but also reduce the system lifetime. A weakness in *MCSoCs*' design is its management, frequently made by a single processor with a single interface with the external world.

The Thesis herein proposed aims to demonstrate that it is possible to develop a distributed *MCSoC* architecture, supporting permanent faults at critical points of the system, as in the processors executing management functions, and at the interface of the *MCSoC* with external entities responsible for deploying new applications into the system.

1.2.4 Objectives

The strategic goal of the Thesis is the specification, development, and validation of methods for fault recovery in manager processors, intending to eliminate the single point of failures in an *MCSoCs*.

The specific goals of the Thesis include:

- Define methods to trigger the fault recovery mechanism after the fault detection;
- Define methods to migrate the memory contents of a faulty *PE* to a healthy one;
- Create a software structure where the manager processors can be recovered by a health *PE*;
- Define the method to connect external hardware modules to the *MCSoCs*;
- Validation of approaches with faults injected in any manager processor.

1.2.5 Original Contributions

The original contribution of this Thesis is a distributed *MCSoC* architecture, with fault recovery capability at critical points in the system. The recovery method includes hardware (Sections 3.1 and 3.2 and Chapter 6) and software modules (Sections 5.3, 5.5 and 5.6), fault monitoring and management recovering. The proposal uses task migration Section 5.5, techniques, and heuristics to select the position of the new manager (Section 5.3). The proposal is scalable, able to act in systems from dozens up to hundreds of processors.

1.2.6 Document Organization

The remaining of this document is organized as follows. Chapter 2 reviews and discusses system management and fault-tolerance related works, positioning the current Thesis with regard to the state-of-the-art. Chapter 3 details the *MCSoC* baseline platform and its evolution in the context of this work. This Chapter presents the two first contributions of the Thesis, the *brNoC* (Section 3.2) and the Injector Module (Section 3.3). Chapter 4 presents a general view of the system management recovery method. This presentation is required to provide a comprehensive perspective of the proposals discussed in the next Chapters. Chapter 5 presents the third contribution of the Thesis, the recovery method of the system management functions, action required when a Manager *PE* becomes faulty. Chapter 6 presents the fourth contribution of the Thesis, the modifications carried-out at the *MCSoC* boundaries to cope with faults during the applications' admission. Chapter 7 presents results related to the methods proposed in Chapters 5 and 6. Finally, Chapter 8 concludes this Thesis, pointing-out directions for future works.

2. STATE OF THE ART

This Chapter reviews and discusses system management and fault-tolerance related works. The Chapter finishes with a comparison between its key features, positioning the current Thesis with regard to the state-of-the-art.

2.1 Johny Paul et al.

Paul et al. [Paul et al., 2015] propose a system management technique. The Authors use a resource-aware computing paradigm called Invasive Computing to reduce the negative effects of resource sharing in MPSoCs with the focus in mobile robotics applications. In the proposal, the operating system (*OS*) can influence the applications' internal decisions, based on dynamic load distribution. Using the resource-aware programming model, the application gains the ability to adapt to available resources by changing its workload.

To achieve the programming model of invasive computing, it is necessary an *OS* specifically designed to support the model. The authors choose the OctoPOS [Schedel et al., 2011] *OS*. This *OS* provides primitives and a scalable and low-overhead execution environment for invasive-parallel applications. Figure 2.1 presents the overall system design of OctoPOS, showing two instances of OctoPOS running on two compute tiles. The proposal has 4 phases: invasive, assort, infect, and retreat. In the invasive phase, applications exclusively acquire resources according to their needs. In the assort phase, the application adapts itself according to the number of resources received from the system. In the infect phase, the processing elements designated to executed the tasks are said "infected", and once the execution has finished, the results can be collected and merged. In the retreat phase, the allocated resources are released.

The Authors executed experiments to evaluate the adaptive algorithm using a platform prototyped in an *FPGA* (Xilinx Virtex-5 XC5VLX330) at 50MHz, with 16 SPARC LEON3 processing elements equally distributed over 4 tiles connected via a custom-designed *NoC*. The authors considered the execution time and cycle-accurate time-stamp counters available in the hardware prototype.

A comparison with two adaptive algorithms, Harris and Shi-Tomasi corner detector, are made to evaluate the new resource-aware algorithm. Applications like audio processing or motor control are used to compare the results. The main conclusions from the work are: (*i*) the model helped to avoid frame drops, no frame was dropped during the evaluation, and the accuracy values improved significantly over the conventional approach; (*ii*) when the resources are the same both approaches have the same accuracy values, but when the resources are scarce the resource-aware model starts to adapt the workload by increasing

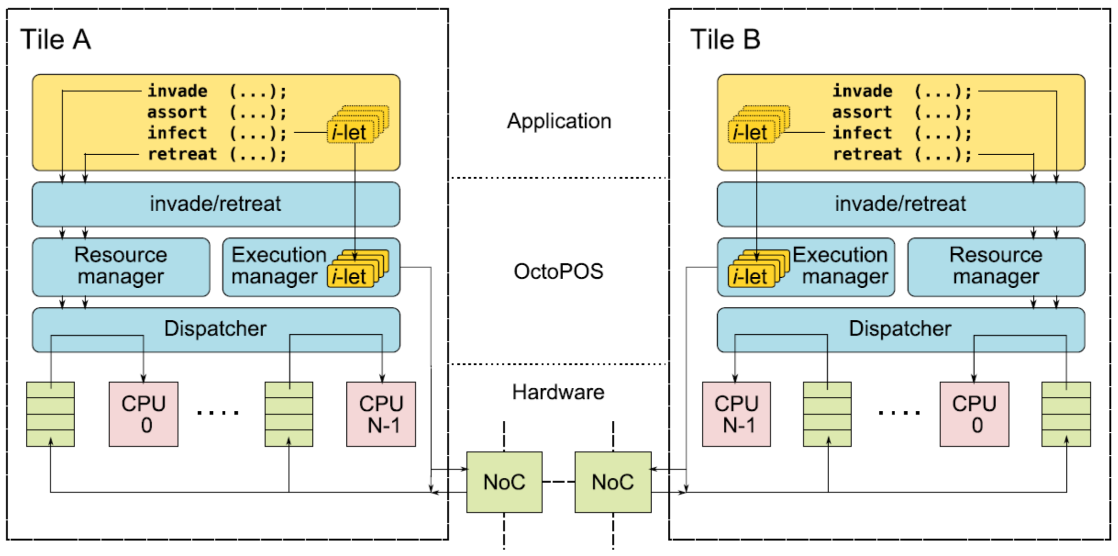


Figure 2.1 – Architectural overview of the invasive hardware/software stack [Paul et al., 2015].

the pruning threshold resulting a slight drop in accuracy however the overall accuracy has improved significantly over the conventional approach; (*iii*) the resource-aware model improves the performance, with up to 22% improvement in the throughput and up to 20% the accuracy.

2.2 André Luís Del Mestre Martins et al.

Martins et al. [Martins et al., 2016] propose a power management method, called Runtime Energy Management (*REM*) to reduce energy while guaranteeing scalability, focusing on homogeneous NoC-based MPSoCs. In the proposal, the power management controls the system by monitoring the energy consumption at the *PE* level using fine-grain *DVFS* as the primary power control policy. Scalability is ensured by a distributed management architecture, responsible for power monitoring and actuation on individual cores to respect power constraints.

To implement the *DVFS* technique in the reference platform, the Authors propose different strategies for modeling the frequency and voltage scaling. To model the frequency scaling, the original *PE* structure was modified, in such a way to have the *NoC* router always running at the nominal frequency, while the processor and memory support different frequencies. To guarantee realistic *DVFS* support, the authors propose a model of voltage scaling considering hardware overheads, as latency and energy.

The *DVFS* actuates on the processor, memory, and *DMNI* as shown in Figure 2.2. With this hardware modification, the processor may work at different voltage-frequency pairs, while the *NoC* transmits packets using the nominal frequency. The *DMNI* module is responsible for synchronizing the modules working at different frequencies. The *DVFS* protocol manages the minimum period for scaling the voltage safely with the frequency range generated by the clock generator.

The *REM* monitors packets sent by each *PE*. All PEs send their energy values to the cluster manager *PE* periodically. The *REM* heuristic defines three energy zones: (*i*) hot zone: energy is above a given high threshold; (*ii*) cold zone: energy is below a low threshold; (*iii*) warm zone: the energy is between hot zone and cold zones. The power zones are configured at the design-time. The manager *PE* receives monitoring packets from the PEs. If a given PE is in the hot zone, a packet is sent to the *SP* to scale down the frequency and voltage; if the PE is in the cold zone a packet is sent to scale up the frequency and voltage.

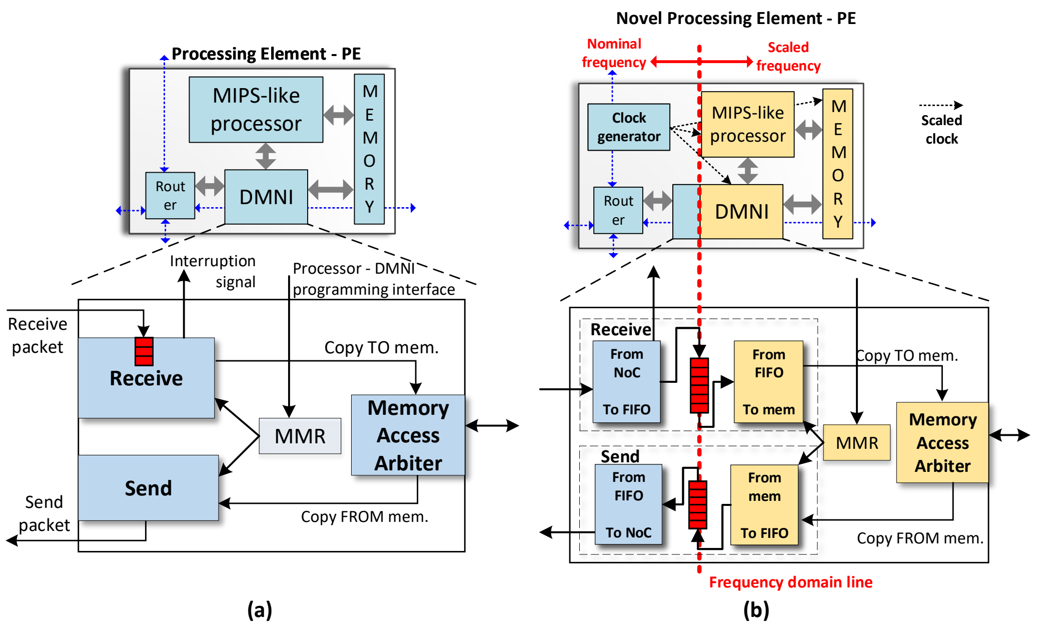


Figure 2.2 – (a) Original *PE* and (b) new *PE* with *DVFS* support. The new *PE* has Clock Generator hardware as well as changes on the *DMNI* [Martins et al., 2016].

Results show an average energy saving of 12.09%, with an execution time overhead of 19.04% to a light power management policy and an average energy saving of 47.39% with an execution time overhead of a 38.22% to a restrict power management policy.

2.3 Paolo Meloni et al.

Meloni et al. [Meloni et al., 2012] propose the MADNESS project, which provides adaptive fault tolerance management in NoC-based MPSoCs. The method uses task migration to handle faults in a processing element that presents a permanent fault by migrating the task to a processor free of faults. The method proposed in [Meloni et al., 2012] involves different layers during the *MPSoC* design. At the application level, a software infrastructure was proposed allowing the execution of the applications in a computational model called Polyhedral Process Network (*PPN*), which consists of autonomous and concurrent processes that communicate with each other through *FIFO* channels. The middleware layer implements support for communication between processes as well as task migration modules for fault tolerance, and the module called Run-time Manager, responsible for making decisions about the resources to be migrated in the presence of a permanent failure in the processing elements. Figure 2.3 presents the software elements to support fault tolerance in MADNESS. The levels presented are:

– Application level, executes the communicating processes;

– Middleware level, implements the communication process layer (*PPN*), the migration process, and the run-time manager;

– Local Operating System, provides basic functionalities such as process management (process creation/deletion, setting process priorities) and multitasking capabilities;

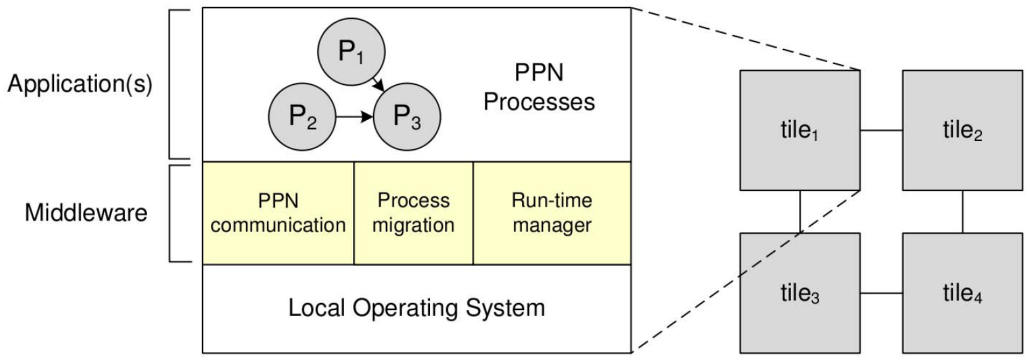


Figure 2.3 – Proposed software stack in the MADNESS project [Meloni et al., 2012].

The last layer developed is at the hardware level, composed of a self-test module used in fault detection and by the task migration process.

The migration module has as main functions: isolate the processor that presented a permanent fault; notify the run-time manager module that it is running on a fault-free processor to execute the task migration; receive all pending messages and *FIFO* tokens relating to predecessor and successor processors (derived from the *PPN* model); and finally send the task context and *FIFO* channels to the triggered run-time manager.

The migration made by the software infrastructure is limited to migrate only the context of the tasks in the failed processor since the task code is loaded on all processors in the system. Figure 2.4 shows the proposed architecture of the self-test module responsible for detecting the permanent faults and signalize to the task migration module to start the process. The task migration hardware disables access to the data and instructions memories of the faulty processing element. However, the *DMA* module has access to the data and instructions memories, enabling to transmit the memory contents even if the *PE* is faulty.

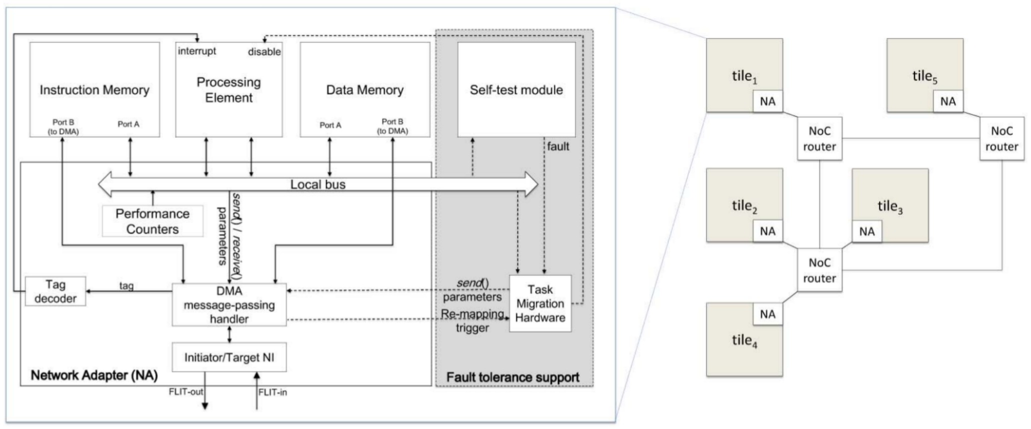


Figure 2.4 – Architecture proposed in the MADNESS project [Meloni et al., 2012].

2.4 Yu-Yin Chen et al

Chen et al. [Chen et al., 2017] propose a fault tolerance management in NoCs. The Authors present a Path-Diversity-Aware Fault-Tolerant Routing (*PDA-FTR*) algorithm for NoCs. The *PDA-FTR* combines path adaptiveness and routing path quality to achieve fault-resilient packet delivery (*FPD*) and traffic load distribution. In the proposal, the algorithm uses the *PDA* information and a local buffer occupancy to acquire the Effective Buffer Length (*EBL*) of the routing direction. *EBL* is a router delay measurement, where higher *EBL* implies shorter routing delay. The routing decision made by the proposed routing algorithm sends the packet to a less congested region and away from the faulty region.

The *PDA-FTR* algorithm, based on the Path Diversity (*PD*) information, initially adopts minimal routing paths for packet delivery. However, as minimal routing paths are blocked by faults, *PDA-FTR* employs non-minimal ones to prevent packet congestion in the faulty region. To ensure the absence of deadlocks, the Authors use the non-minimal Odd-Even turn model [Tsai et al., 2013]. If there are two or more routing path candidates, the *PDA-FTR* uses the *EBL* information of each candidate channel to select the better output channel.

The Authors report that to implement the *PDA-FTR*, it is necessary to store the *FPD* (Fault-Location-Based PD) table at every router and the entire *FPD* table in a router. The number of tables entries in a router increases by $O(k^2)$, where k is the number of routers. This feature compromises scalability, and the computing time and power consumption grow when searching for a specific entry in a large table. To reduce the memory cost, the Authors propose a regional *FPD* table to cover most routing paths for packet transmission while minimizing performance degradation. Due to the data locality, processing elements that often communicate with each other are usually placed in proximity to each other for minimizing the delay in data delivery. Based on this property, a small table can store the most used paths for making a routing decision with minimal performance degradation when compared to full *FPD* table.

Results show that the router with *PDA-FTR* has an area overhead of 10.77% compared to the baseline design. Compared with the baseline design router, the proposed *PDA-FTR* router has power overhead of 7.04% due to the routing table.

2.5 Arezoo Kamran et al.

Kamran et al. [Kamran et al., 2016] propose an autonomous test mechanism for online detection of permanent faults in many-core processors. In this method, several test components are incorporated in the many-core architecture that autonomously and concurrent with the system normal operation, distribute software-based self-test routines among the processing cores, monitor the behavior of the processing cores during the execution of the test routines, detect faulty cores, and make their suppression from the system if possible. To use short idle times of the processing cores, test data is segmented into small pieces, called test snippets. Individual test snippets are distributed among the processing cores and are made accessible to them for a limited period. If a processing core has an idle slot during a period that a test snippet is available, it executes the test snippet, otherwise, it skips execution of that portion of the test. The proposed test mechanism is designed in such a way that it supports skipping of test snippets at the expense of losing test quality, but without any effect on the integrity of the whole test mechanism.

The Authors assume that the many-core processor contains identical nodes, each of which with a processing core, cache blocks, and hardware facilities for communications with other nodes. The test architecture is independent of core communication and uses a dedicated test distribution logic. The processing cores are tested using a non-intrusive software-based self-test approach (*SBST*). In the proposed technique, a small amount of assisting hardware is incorporated in the many-core architecture. These hardware test components distribute *SBST* among the processing cores and detect idle processing cores to switch them to test mode. The *SBST* routines execute, monitoring the behavior of the processing cores during the test.

Figure 2.5 shows the hardware components added near a processing core. Several adjacent processing cores share this additional test component. A local test controller, called *cluster tester*, receives a test routine from a shared global controller called Autonomous Chip Tester (*ACT*), and stores it in a small local buffer (called test-snippet buffer). When this cluster tester is triggered by a command from the *ACT*, it starts monitoring the processing cores. When a processing core becomes idle, the cluster tester disconnects the idle processing core from the communication infrastructure and the memory subsystem and connects it to test-snippet buffer while the other neighboring processing cores are performing their normal

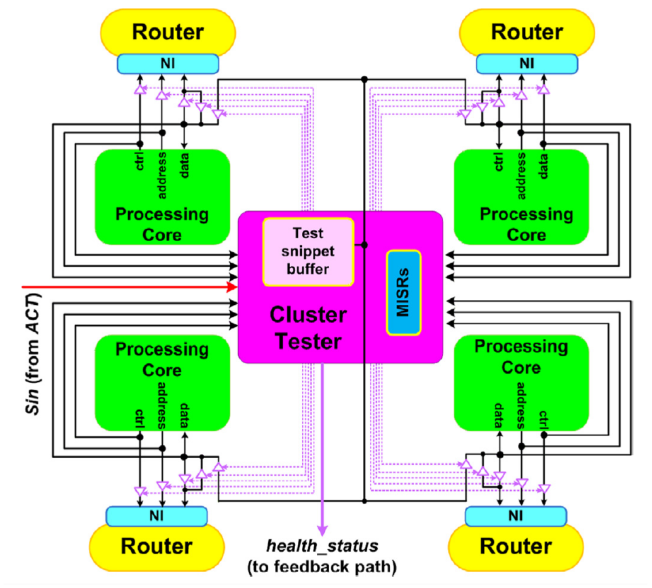


Figure 2.5 – Hardware test components placed near the processing cores [Kamran et al., 2016].

operation. Meanwhile, the disconnected processing core starts executing the test routine execution, the cluster tester captures bus activities of the processing core under-test, and generates a signature. This signature is used later to be compared with the signatures generated by other processing cores or with a golden signature. The responsibility of a cluster tester is to receive test commands and test data from the *ACT*, detect and isolate idle processing cores of the cluster, apply test data to the idle processing cores, and identify and remove faulty processing cores in collaboration with the *ACT*.

Experimental results show that, for a cluster with four Plasma MIPS processing cores, there is a hardware overhead of 3.6%, and for larger cluster sizes, as 32 Plasma MIPS, 1.2%. The authors related that there is no performance overhead because the test executes when the processing cores are in idle status.

2.6 Biswajit Bhowmik et al.

Bhowmik et al. [Bhowmik et al., 2016] present a distributed online test mechanism that detects stuck-at faults (*SAF*s) in the *NoC* channels as well as identifies the faulty channels. The proposed test mechanism improves yield and reliability of NoCs at the cost of a small performance degradation. The method focus on detecting stuck-at-0 (*SA*0) and stuck-at-1 (*SA*1) faults in the channels and evaluate their effect on network performance. Each channel consists of control, data, and handshake wires. The authors propose an on-line method for testing wires connecting the router to its core.

The proposal assumes a 16-bit channel. The method starts the application of the test on a 2 × 2 *NoC* configuring the channels as unidirectional links. Thus, there are 256 wires (8 channels between routers and 8 channels between routers and cores). The proposed test model tests 256 *SA*0 and 256 *SA*1 faults. The test technique accounts these faults on data, control, and handshake wires. The method test a channel in this sequence. During the test of a channel, the method keeps the underlying network functional, except the subset of channels under test in a test iteration. In this mode, the network is allowed to transmit application data but must wait at an intermediate router involved in the test. The algorithm uses test packets that contain the test vectors. The packet contents vary depending on the type of wire under test. That means stuck-at faults in data, control, and handshake wires get detected with a specific packet format. To test data wires, test vectors are placed in the payload field of the test packet. After transmitting the test packets, the test response analyzes at receiver core and routers analyzes received test vectors to detect whether data, channel, and wire experiences any *SA*0 or *SA*1 fault. If the wire experiences an *SA*0 or *SA*1, the test response analyzes receives logic-0 or logic-1 on the wire. The detection ensures the state of the faultiness of a wire. After testing the data-wires, the control and handshake wires are undergone the testing similarly.

The Authors show that during the experiment they observe that the Test Packt Generator takes one clock to generate a test packet. Another clock cycle is used to organize it as a test packet. The packet takes four clocks to reach the neighbor and additional two clocks to analyze the received test sequences. Thus, a channel can be tested in just 11 clocks. Multiple channels in an iteration need this time to be tested. The authors related that the link cover metric in the proposed test mechanism achieves 100%.

2.7 Cristiana Bolchini et al.

Bolchini et al. [Bolchini et al., 2013] describe a system with an adaptive level of reliability. The work presents a fault management layer at the *OS* level. This layer has a strategy for dynamically adapting the reliability at run-time. The fault management layer contains three methods: duplication with comparison (*DWC*), triplication (*TMR*), duplication with comparison and re-execution (*DWCR*).

The *DWC* technique guarantees the fault detection property by creating a replica of the application and by comparing the outputs. A checker task is issued at the end of each node of the application's task graph to identify discrepancies in the (intermediate) results.

The *TMR* technique creates two replicas of the original application, resulting in three results to be voted by specific 2-of-3 majority voter that mitigate the possible occurrence of faults. Besides the fault tolerance property, the technique is also able to achieve fault diagnosis features, by identifying the core producing the erroneous mismatching value.

The *DWCR* similarly to the *DWC* technique, the original application is duplicated to have the possibility to detect possible faults by comparing two results using a checker task. If an fault is detected, a third replica of the task is created and executed. In this way, a voter task can identify the correct result. This technique provides the fault tolerance property, and may provide fault diagnosis features, i.e., it can identify the core that caused the fault. This technique is characterized by a limited overhead for achieving the fault tolerance property because the third replica is used and scheduled only after a problem has been detected.

The method uses an observe-decide-act (*ODA*) control loop as shown in Figure 2.6. The *ODA* loop is divided into three stages. The observe phase consists of sensing the status of the system and, in particular, collecting execution data for computing a set of performance and reliability-related metrics. Then, the decide phase is performed by considering the measured metrics and a high-level goal specified as a requirement on the application (as performance and reliability). The knowledge of the goal guides the adaptation engine in making a suitable decision on how to execute the applications. Finally, once the decision has been taken, it is put into practice in the act phase through the actuators, which modify the system's knobs to alter its behavior. The behavior of the system is sensed again in the observed phase, and the control loop is restarted.

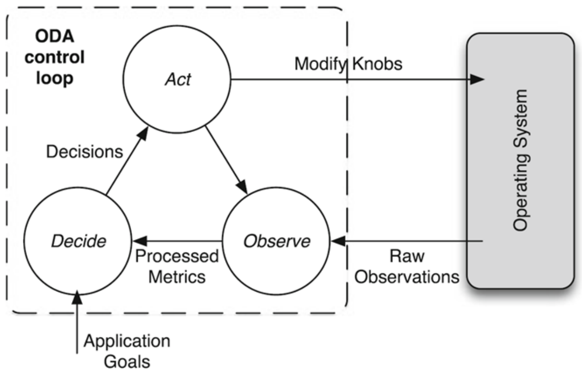


Figure 2.6 – The Observe–Decide–Act control loop of [Bolchini et al., 2013].

If the method experiences a high detected fault ratio, *DWCR* is highly disadvantageous compared to *TMR* since it would require a considerable number of tasks to be re-executed. The last knob is the resource activation/deactivation. The fault management techniques, during the execution, the adaptation engine may diagnose a suspected damaged processing core. In this case, the engine can deactivate the processing core to further analyze it using specific diagnosis tasks. Later, if the result of the accurate analysis is negative, the unit can be reactivated and used again for executing the application.

2.8 Tsoutsouras et al.

Tsoutsouras et al. [Tsoutsouras et al., 2017] present a run-time resource management framework which can dynamically adapt the system to permanent faults in a self-organized, workload-aware manner. They proposed an organization that allows resource management agents to recovery from a failure electing a new agent to replace the faulty management agent, while workload awareness optimizes the election according to the status of each core.

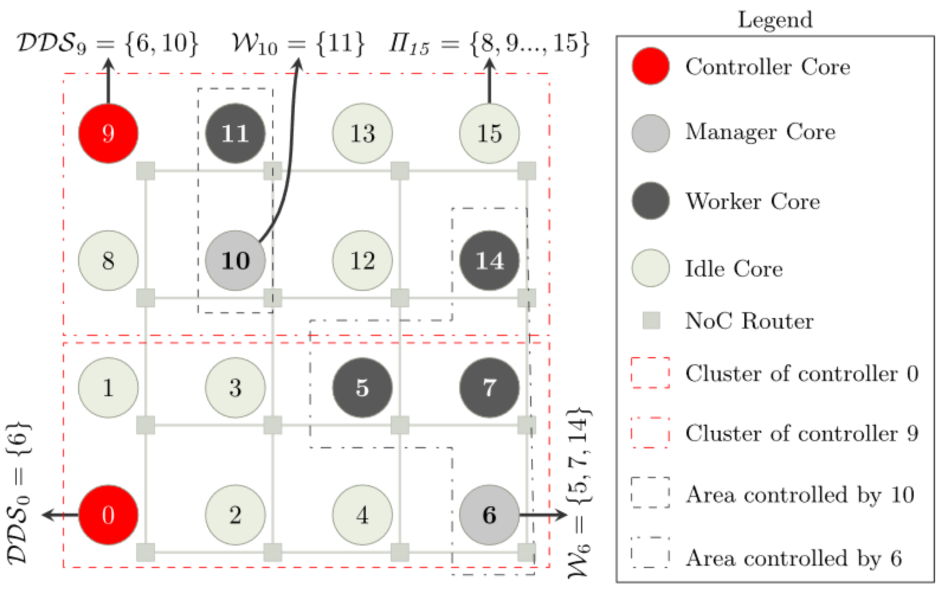


Figure 2.7 – Overview of SoftRM proposal [Tsoutsouras et al., 2017].

The work is hierarchically organized as shown in Figure 2.7:

- *Controller* cores (red in Figure 2.7). Responsible for monitoring the system status. There is no central point of system monitoring. Each Controller core is dedicated to its cluster and it is not involved in application management or workload execution. Each cluster area of the system is monitored by a controller core. These clusters are not overlapping and their number and topology is parameterizable and can be defined at the system initialization, but cannot change at run-time;
- *Manager* cores (dark gray in Figure 2.7). The relationship between a manager core and an application is one to one, meaning that there is one manager core per application. This one to one relationship allows that each manager core adopt different workload allocation schemes, without any resource sharing between applications;
- *Worker* cores. Execute the applications' tasks. Each worker core has a controller and a manager core. This design choice decouples system monitoring and application management. Cores without a manager and a controller core are considered Idle Cores.

The technique allows fault tolerance recovery to any agent in the system (controller, manager and worker). When a fault is detected, an election occurs to determine which *PE* replace the failed agent. The *PE* with the lowest workload is elected to replace the failed agent. After the election, the state of the system is updated and the failed *PE* is ”removed” from the system.

The SoftRM was implemented and evaluated on Intel Single-chip Cloud Computer (*SCC*) *NoC* based many-core system. The *SCC* chip consists of 24 dual-IA-core tiles connected by a 2D-grid on-die network, where each tile contains two P54C cores.

2.9 Suraj Paul et al.

Suraj Paul et al. [Suraj Paul, 2018] present a fault-tolerant resource allocation strategy to mitigate the effect of permanent faults on processors, targeting mixed critical applications. The goal is to select a suitable fault tolerance strategy to mitigate the effect of processor failure at run time with minimum degradation in the performance of the executing applications.

Figure 2.8 presents the system model. A special purpose *PE* hosts the real-time operating system (*RTOS*), and it is referred as Manager Core. The work assumes that such PE acting as Manager Core is reliable and faulty-free. The Manager Core executes task allocation. When a user submits an application to the system, its tasks are stored in the Task Memory. The status of every PE, i.e., faulty/non-faulty, is updated in the resource manager. The Task Mapping and Scheduling Unit (*TMSU*) has two sub-units, Task Mapper (*TM*) and Task Scheduler (*TS*). The allocation algorithm is present in *TM*, which is executed at runtime for assignment of tasks to different available PEs. *TS* schedules the allocated tasks depending on their timing characteristics and criticality.

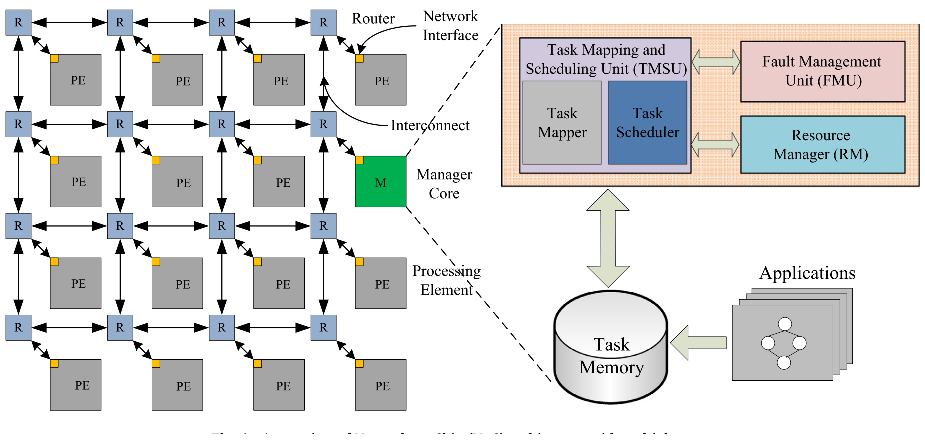
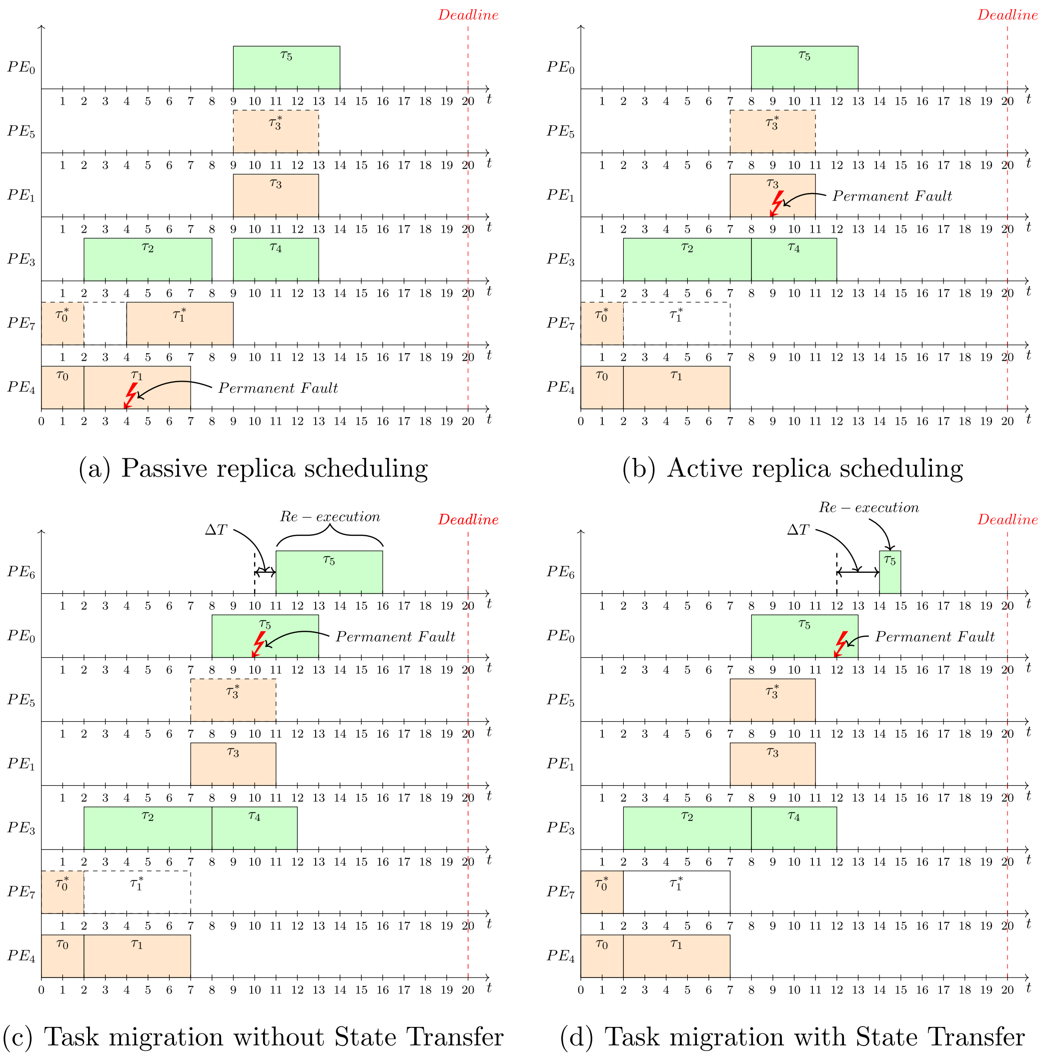


Figure 2.8 – A Overview of Network-on-Chip (NoC) architecture with multiple cores [Suraj Paul, 2018].



(a) Passive replica scheduling (b) Active replica scheduling

(c) Task migration without State Transfer (d) Task migration with State Transfer

Figure 2.9 – Fault Tolerance using replica tasks [Suraj Paul, 2018].

Figure 2.9 presents four different scenarios using the proposed fault-tolerant algorithm. Note the τ0, τ1 and τ3 are critical tasks, so a replica from these tasks are created in different PEs.

- Passive replica scheduling – Figure 2.9(a). In this scenario, PE_4 became faulty at time t=4. To overcome the effect of this fault on executing task τ1, its passive replica, τ1* allocated on PE_7, is scheduled.

- Active replica scheduling – Figure 2.9(b). A permanent fault affects the execution of task τ3 mapped on PE_1 at t = 9. Since this is a critical task with a low slack time, the replica task τ3* is already scheduled concurrently with τ3, which tolerates failure of PE_1.

- Task migration without State Transfer – Figure 2.9(c). PE_0 is assumed to become faulty at t = 10, while executing task τ5. As the fault occurred close to the start time of the task, the proposed fault mitigation policy re-executes the task on an alternative PE, PE_6.

– Task migration with State Transfer – Figure 2.9(d). PE_0 fails at t = 12. A different fault tolerant strategy is adopted, which helps to complete the task within its deadline. It can be seen that in this case the fault occurs close to the task $\tau 5$ deadline. Therefore, state transfer policy is used to recover from fault. Here, the state of the task $\tau 5$ consisting of both code and data is transferred to the nearest available PE_6 and the task resumes its execution.

The fault-tolerance method effectiveness of dynamic mapping and scheduling algorithm was implemented on a simulator developed in C++ and used in ORION 3.0. Scalability, *QoS* performance, communication latency and communication energy was evaluated using the ORION 3.0 model. The proposed algorithm uses an unified mapping and scheduling strategy that gives an energy aware resource allocation in both fault-free and faulty scenarios.

2.10 Domingues et al.

[Domingues et al., 2018] propose a system management technique targeting communication between PEs. The Authors proposer a lightweight fault recovery mechanism for brokers of a publish-subscribe middleware for MPSoCs. The proposed approach uses the existing brokers to backup sensitive data of its neighbor brokers, which provides high availability to the system because when a fault is detected in a broker's processor, its neighbor broker promptly assumes the responsibility of managing the applications of the faulty broker. This broker replacement is entirely transparent to the application level.

Figure 2.10 illustrates a case study using a 6x6 NoC-based MPSoC, with 3x3 clusters. The proposed fault recovery approach relies on a 3-stage protocol consisting of monitoring, cluster recovery and broker recovery phases. The arrows in the figure indicate that the broker at the end of the arrow (called primary broker) is monitored by the broker at the start of the arrow (called secondary broker). This ring topology is reconfigured in the presence of a faulty broker.

The broker fault tolerante feature has the following configurations that must be defined at design-time: (a) the time span between keepalive requests (30,000 clock cycles in the performed experiments); (b) the number of unanswered keepalive requests to consider the broker as faulty (3 in performed experiments).

The MPSoC hardware infrastructure was described using OVPSIM APIs by Imperas, which provides an instruction accurate simulation framework. The kernel software was implemented in C programming language and the middleware software using the C++ programming language.

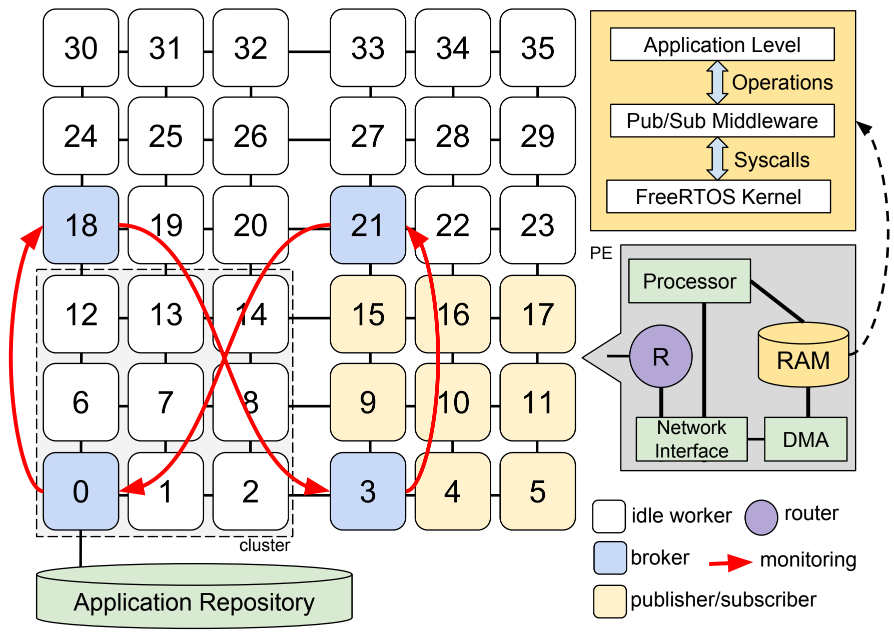


Figure 2.10 – A 6x6 platform with 3x3 clusters configuration [Domingues et al., 2018].

The results presents the broker recovery time (*BRT*) is in average 32,000 clock cycles. Results showed the approach had a minimal resource overhead for different fault insertion setups.

2.11 Silveira et al.

[Silveira et al., 2016] proposes a technique that employs the preprocessing of fault scenarios based on forecasting fault tendencies, which is performed with a fault threshold circuit operating in accordance with high-level software.

The work focuses in a reconfigurable fault-tolerant system for irregular networks that requires the follow mechanisms for (*i*) fault detection and diagnosis; (*ii*) fault recognition reporting; (*iii*) deadlock-free routing computation; and (*iv*) routing reconfiguration (e.g., routing tables and auxiliary circuits).

The OsPhoenix's Kernel contais (Figure 2.11): (*i*) the Control Module that manages the fault-tolerant mechanism; (*ii*) the NoC Driver. The Global Fault Table is a module that stores the status of fault tendency for all the NoC links. The Control Module update the Global Fault Table and synchronize the OsPhoenix knowledge with all the PEs. The Scenarios Processing Module is responsible to compute the routing tables managed by the Control Module using the information provided from the Global Fault Table. When a new routing path is discovery it is stored and can be transmitted to the HwPhoenix.

The Authors related that the preprocessed scenarios reduces the time that the NoC is halted. The amount of scenarios grows exponentially with the quantity of faults.

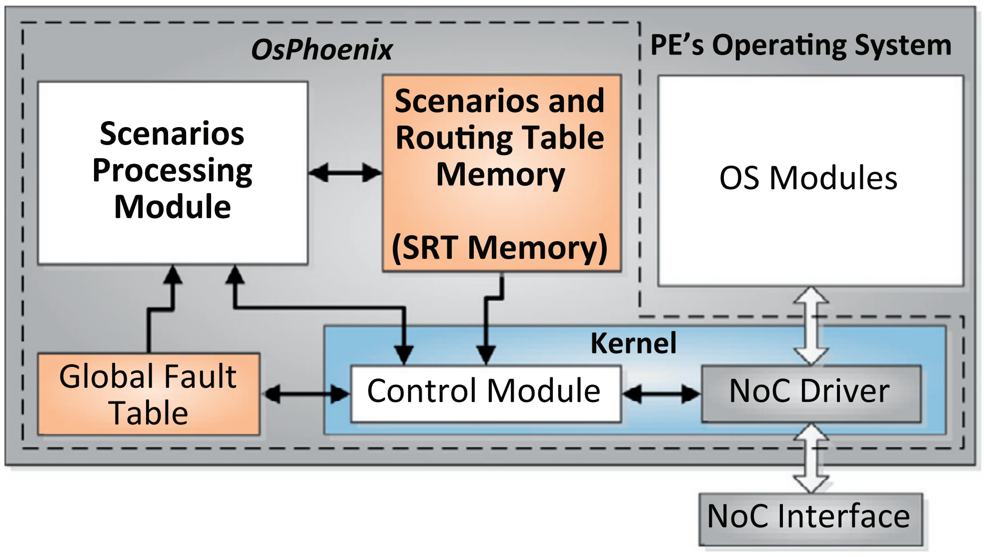


Figure 2.11 – Block diagram of the OsPhoenix architecture [Silveira et al., 2016].

That requires and overhead of area to compute all scenarios. To minimize this problem the work proposes a differential treatment for static and transient faults, an incremental processing of fault scenarios and dissimilarity approach.

2.12 Related Work Analysis

Table 2.1 summarizes the reviewed works according to the classification chosen for system management and fault-tolerance comparison. The first column contains the Author and reference. The second column shows the constraints applied to the system, which is management or fault-tolerance. The third column presents the architecture (homogeneous, heterogeneous or only *NoC*) and the core counting given by the number of processor elements or the *NoC* size. The fourth column is the method or technique main goals under the constraint (second column). The fifth column lists the techniques used to control the system according to the Authors definitions. The last column presents the experimental setup used by the Authors and the abstraction level of the system modeling which the results are produced according to the following standard: (*i*) cycle-accurate simulation, only the execution time result is exact, and the others are estimated; (*ii*) *FPGA* prototyping, cycle-accurate simulation for *FPGA* devices.

The literature presents distinct management and fault-tolerance approaches for many-core systems that can be applied to the PE modules. According to the Table 2.1, several techniques are used to manage the system with different goals, as power, resources, and performance. Fault-tolerance may be applied at different levels, as routing-algorithm, link level, processor level, system level.

The literature presents a rich fault tolerance and management approaches for *MCSoCs*. However solutions that encompass a fault tolerance focused on the system management are scarce. [Tsoutsouras et al., 2017] is the work that most similar with this Thesis proposal. The main difference is that the [Tsoutsouras et al., 2017] implemented the manager recovery in different hierarchical levels. They have tree levels of management and recover, the first level is the replacement of the Controller cluster. The second level is the recover from the Manager PE (manager from an unique application) and an worker recovery (PE that execute an task). To execute the recovery method in [Tsoutsouras et al., 2017] they have a communication protocol to update and leave the system in a safe state. In this work we assume that the memory is protected, similar to [Meloni et al., 2012], and a hardware module in the fault manager handle the migration of data and contents to a health PE.

The proposal made in this Thesis includes fault tolerance techniques at the *MCSoC* management level, including the local management in the clusters and the global system management (Chapter 5). Fault tolerance is also proposed for the admission of new applications in the *MCSoC*, through the redundancy of the links with external devices, as well as redundancy with external devices (Chapter 6). Thus, the proposal is original and advances the state-of-the-art in fault tolerance for MCSoCs, through the proposition of high-level techniques, given that low-level techniques (router and processor) are mature in the literature.

Table 2.1 – Summary of the the state-of-the art.

Author	**Design Constraint**	**Architecture** **# of cores**	**Design Goals**	**Techniques**	**Modelling** **(Exp. Setup tools)**
[Paul et al., 2015]	Resource Management	Heterogeneous, 16 SPARC LEON3	Performance	Dynamic load distribution, adaptive shared resources	FPGA Simulation (Xilinx Virtex-5, XC5VLX330 FPGA)
[Martins et al., 2016]	Power Management	Homogeneous, 3x3 to 12x12	Scalability and energy efficiency	DVFS, clock gating, mapping, migration	Cycle-accurate simulation, and low-level analysis (Cadence tools)
[Meloni et al., 2012]	Fault-Tolerance on processor cores	Homogeneous 2x2	Task remapping	Task migration	FPGA simulation, Design Space Exploration (DSE)
[Chen et al., 2017]	Fault-Tolerance on Routing algorithm	Only NoC, 8x8	Adaptive routing algorithm, balance traffic load	Path discovery	Cycle-accurate simulation
[Kamran et al., 2016]	Fault-Tolerance on processor cores	Homogeneneous, 2 to 32 processing cores	CPU faulty detect	Software-based self-test routines	Xilinx ISEs WebPACK simulation
[Bhowmik et al., 2016]	Fault-Tolerance on links	Only NoC, 2x2 to 8x8	On-line test mechanism that detects stuck-at faults	Packet test	Xilinx 10.1 simulation
[Bolchini et al., 2013]	Management and Fault-Tolerance on processing cores	Homogeneous, 6 to 12 processing cores	Performance, detected faults	Application's replicas	Cycle-accurate simulation + ReSP simulation environment
[Domingues et al., 2018]	Fault Tolerance in Management Cores	Homogeneous 2 to 36	Fault Detection and Manager Recovery	Migration of data contents and cluster reconfiguration	OVPSIM, instruction-accurate simulation
[Tsoutsouras et al., 2017]	Fault Tolerance in Management Cores	Homogeneous 6 to 24 PE	Fault Detection and Manager Recovery	Replacement Core	Intel Single-chip Cloud Computer (SCC)
[Suraj Paul, 2018]	Fault Tolerance in Processor cores	Homogeneous 4x4 to 10x10	Fault Tolerant on critical applications	Task replicas	ORION 3.0 Simulator
[Silveira et al., 2016]	Fault Tolerance in Data NoC routers	Homogeneous 2x2 to 8x8	Fault Detection and routing reconfiguration	Deadlock-free routing computation	Cycle-accurate simulation
This Thesis	Fault-Tolerance on processing cores	Homogeneous, parameterizable size	System Management Recovery	Kernel manager mapping, task migration	Cycle-accurate simulation

3. BASELINE PLATFORM

This Chapter introduces the *MCSoC* baseline platform used in this work, based on the Hermes MultiProcessor System (*HeMPS*) [Carara et al., 2009, Woszezenki, 2007] and its evolution in Section 3.1. Section 3.2 presents the control *NoC* (*brNoC*) and Section 3.3 the Injector Module. The baseline platform and this work are both developed at the *Hardware Design Support Group* (*GAPH*) research group [GAPH, 2018].

The *brNoC* (Section 3.2) and the Injector Module (Section 3.3) corresponds to the first two contributions of this Thesis.

3.1 Baseline Platform and its Evolution

Figure 3.1 presents the reference baseline platform. The architecture contains a set of *PE*s interconnected by a data *NoC*. The Global Manager (*GM*) has an interface with the external environment to the *MCSoC* to receive new applications. Each *PE* contains one processor (32 bits MIPS), a Direct Memory Network Interface (*DMNI*, combining the functions of a Network Interface and a *DMA* module) [Ruaro et al., 2016]. The PEs' hardware is the same, being the role assigned to the *PE*s made by software: Slave *PE*s (*SP*s) execute users' tasks, supporting multitasking and message exchanging; *GM*/*LM* manage a given cluster.

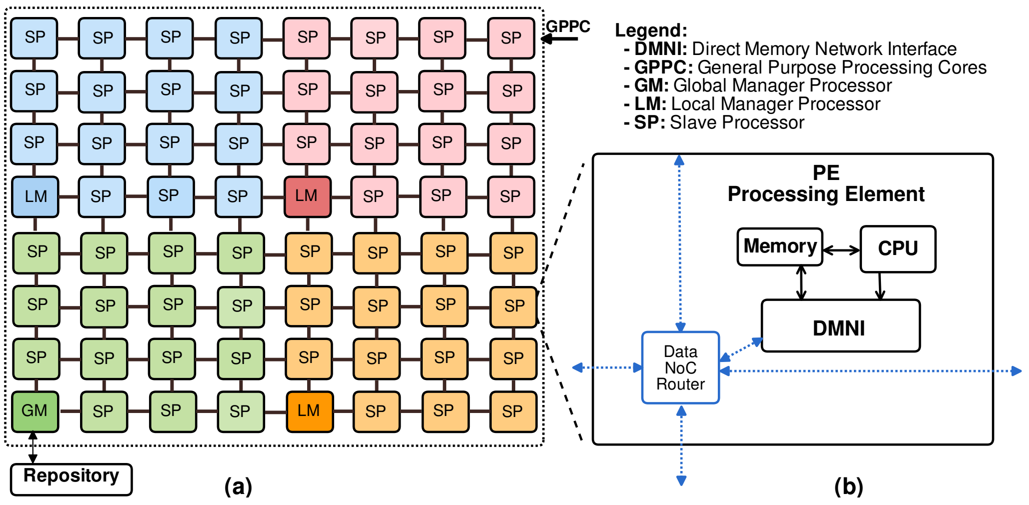

GPPC
Legend:
- DMNI: Direct Memory Network Interface
- GPPC: General Purpose Processing Cores
- GM: Global Manager Processor
- LM: Local Manager Processor
- SP: Slave Processor
PE
Processing Element
Memory
CPU
DMNI
Data NoC Router
SP
LM
GM
Repository
(a)
(b)

Figure 3.1 – HeMPS baseline architecture.

The main baseline architecture features include:

- Global Manager attached to the Application Repository;

- True-dual local scratchpad memory, with one port connected to the *CPU*, and the second one to the *DMNI*;
- Single 32-bit physical links;
- Wormhole packet switching;
- Suport to XY routing algorithm;
- Input buffer depth: 8 flits.

Figure 3.2 presents the evolution of the baseline platform carried out in this Thesis. The main differences include: (*i*) addition of a Control *NoC* (Section 3.2); (*ii*) control wrappers; (*iii*) Application Injector *IP* core and support for peripherals (Section 3.3); (*iv*) duplicated physical channels. Two similar descriptions model the platform: synthesizable *VHDL*, for characterization purposes; SystemC at the *RTL* level, with clock-cycle accuracy, enabling the simulation of systems with dozens of *PE*s.

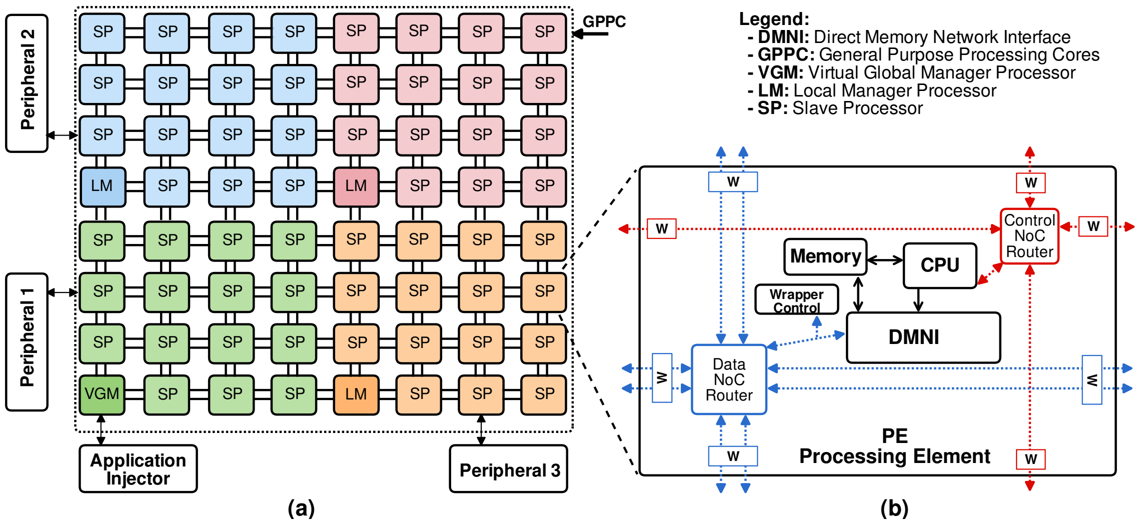


Figure 3.2 – Modified architecture adopted in this Thesis.

The data *NoC* main features includes: (*i*) 2-D mesh topology; (*ii*) 8-flit buffer depth, input buffering; (*iii*) wormhole packet-switching; (*iv*) support for deterministic XY and source routing; (*v*) credit-based flow control; (*vi*) duplicated physical channels (two 16-bit channels per link), enabling full adaptive routing.

As can be observed, the most significant changes in the data *NoC* correspond to the links' duplication (16-bit links were used to avoid area overhead), resulting in two 16-bit disjoint networks; and support for source routing. These two modifications are important in the context of fault tolerance, because it allows applying dual routing algorithms (for example, west-first and east-first) at each subnet, and thus ensure fully adaptive routing, free of deadlock.

The control *NoC*, detailed in Section 3.2, transfers the control messages. This work uses the control *NoC* to transmit messages with the following purposes:

- notify the status of the *VGM*/*LM*;
- notify the new application admission to *VGM*;
- freeze application(s) managed by a given Manager;
- notify an *SP* that it will become a new Manager;
- notify a *DMNI* module to transfer the memory contents of an SP to a new system address;
- notity the *SP* about a new Manager address;
- unfreeze application(s) after the Manager migration.
- notify an *SP* to discard an incomplete task.

A new architecture feature is that the memory is accessible by the data *NoC* even if the processor has a permanent fault. The *control NoC* configures the *DMNI* module to transfer the memory contents to another PE. This feature, transfer the memory contents when the processor has a permanent fault, is commonly adopted in fault-tolerant approaches [Meloni et al., 2012].

The methods proposed in this Thesis may be applied to homogeneous or heterogeneous *MCSoC*. The proposed method requires the following architectural features: (*i*) a set of *PE* with the same architecture; (*ii*) at least two disjoint *NoC*, one for application data and one for management purposes [Wentzlaff et al., 2007]; (*iii*) a memory module that can be read/write directly by the network interface [Meloni et al., 2012].

Briefly, the changes made by the Author in the reference architecture include:

- *DMNI* received two modules: send kernel and receive kernel, and started to serialize the data incoming from the *CPU* (32 to 16 bits) and to parallelize the data coming from the data *NoC* (from 16 to 32 bits);
- Added wrappers in the control signals of both NoCs;
- Data *NoC*: support to 16-bit flits (buffer, switch control, crossbar), support to XY and source routing, duplicated physical channels;
- Control *NoC*: broadcast as the default transmission mode, non-intrusiveness (i.e., decoupled from the Data *NoC*), full reachability (the broadcast mode ensures that if there is a path to a given router, the path is found);
- Inclusion of the support for peripherals, such as the Application Injector (Section 3.3).

3.1.1 Data NoC

The data *NoC* transfers *data messages*, exchanged by applications. The data *NoC* extends the *NoC* Hermes [Moraes et al., 2004] adopting duplicated physical channels, flit width equal to 16 bits, input buffering, round-robin arbitration, credit-based flow control, wormhole packet switching, simultaneous support for distributed XY routing and source routing (*SR*).

The use of duplicated physical channels ensures deadlock avoidance and full routing adaptivity. 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 [Linder and Harden, 1991]. The flit width is half of the original in the Hermes *NoC* to minimize the area overhead due the duplicated physical channel adoption.

The standard routing mode between PEs is the distributed XY routing algorithm. The data *NoC* also supports source routing (*SR*) such that it is possible to determine alternative paths to circumvent faulty routers. The mechanism to found an alternative path to use in the *SR* is presented in Section 3.2.

The reference *MCSoC* architecture, *HeMPS*, does not support the communication with peripherals. This platform was modified in such a way to support such hardware components. Section 3.3 presents an example of an external device -*Application Injector*. The data *NoC* differentiates *data* packets from *peripheral* packets. Data packets are those exchanged by tasks running in PEs, and peripheral packets are those transferred between a task and a peripheral. A *peripheral* packet arriving in a boundary *PE* goes to the peripheral, and not to the *DMNI*.

A *data* packet, from the *NoC* point of view, has a header and a payload (Figure 3.3). The packet header content is used to control the data *NoC* behavior, such as, routing and arbitration. While in [Carara et al., 2009] the packet header have two fields (target and payload size), we adopt three fields to support the *SR*, the rerouting mechanism and the communication with peripherals: (*i*) the source/target address with data (D) or peripheral (P) packet flag; (*ii*) the XY or *SR* field that indicates the turns on each router when use SR or the source/target address when use XY routing (repeat the first field) and; (*iii*) the payload size.

From the task point of view, a message is used by kernel with two fields: (*i*) the message header to control the data exchange between tasks or with peripherals through data such as, producer task ID, consumer task ID, service (e.g. message delivery, request for a message, task mapping, task allocation), message timestamp and, (*ii*) the payload, an optional field, with the task or peripheral data. It may contain, for example, user data or the object code of a task.

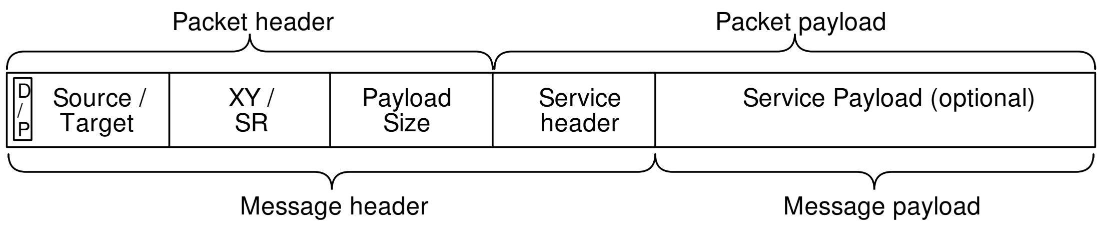


Figure 3.3 – Packet and message structures - a flag (D/P) in the target address field differentiates *data* packets from *peripheral* packets.

3.1.2 Software Model

Scalability at the hardware level comes from PEs executing several tasks in parallel, using the *NoC* to transmit concurrently multiple flows. However, large systems require high-level management for controlling the deployment of new applications, monitoring resources usage, manage task mapping and migration, and can execute self-adaptive actions according to systems constraints. Thus, to achieve a scalable design, *HeMPS* adopts cluster-based decentralized management [Castilhos et al., 2013]. Clusters are virtual regions, with a set of slave processors (S_{PE}) and one manager PE (M_{PE}). S_{PE}s execute applications' tasks, while M_{PE}s manage the clusters.

The management occurs at the M_{PE} and S_{PE} levels, executed by the kernel running in those PEs, as depicted in Figure 3.4.

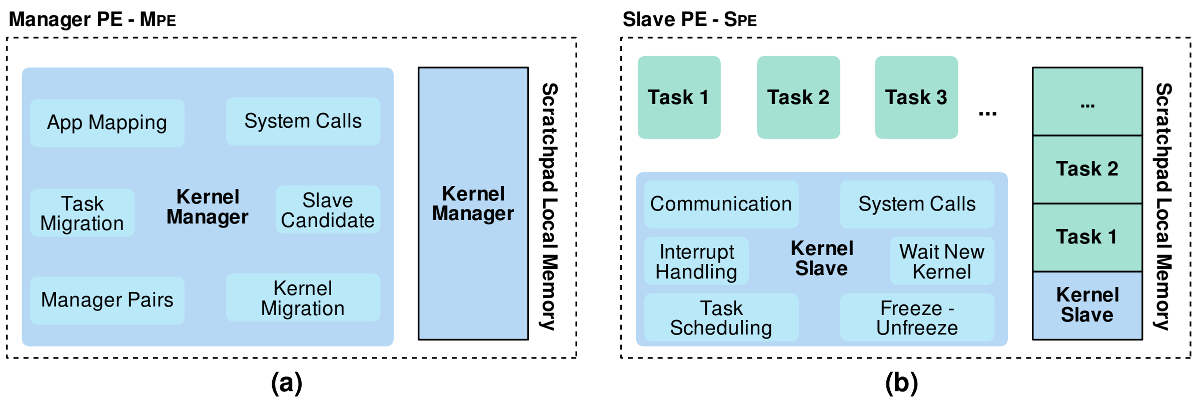


Figure 3.4 – Overview of the kernels: (a) M_{PE} kernel manages the system and do not execute users' tasks; (b) S_{PE} kernel manage users' tasks.

At the M_{PE} level, Figure 3.4.a, the local memory is reserved to the kernel, without executing user's tasks. The M_{PE} executes heuristics as task mapping, task migration, monitoring, kernel migration, manager pairs, slave candidate.

At the S_{PE} level, Figure 3.4.b, a multi-task kernel acts as a operating system. The platform adopts a paged memory scheme to simplify the kernel design. Examples of actions executed by the kernel include task scheduling, inter-task communication (message passing), interrupt handling, freeze, unfreeze, wait new kernel.

Both manager kernels are written in C language. Only a small part of the code is written in assembly language, responsible for executing context saving and handling hardware and software interruptions.

Applications are also written in C language. They are modeled as task graphs A $= < T, P, D >$, where T = $\{t_1, t_2, ..., t_m\}$ is the set of application tasks corresponding to the graph vertices, P = $\{p_1, p_2, ..., p_n\}$ is the set of peripherals corresponding to the graph vertices. The D set represents the application descriptor which contains the communicating pairs $\{(t_i, t_j), (t_i, p_r), (t_j, p_s), ..., (t_m, p_n)\}$ with $(t_i, t_j, ..., t_m) \in$ T, $(p_1, p_2, ..., p_n) \in$ P. A pair (t_i, t_j) denotes the communication from task t_i to task t_j $(t_i \rightarrow t_j)$, and a pair (t_i, p_r) denotes the communication from task t_i to peripheral p_r $(t_i \rightarrow p_r)$. Figure 3.5 present an application modeled as task graph.

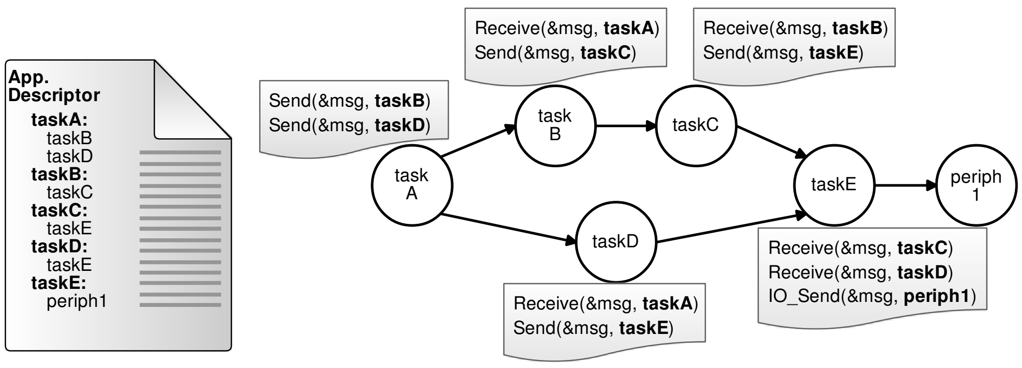


Figure 3.5 – Application task graph example.

Tasks communicate using message passing (MPI-like) primitives. The *API* provides two 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. The *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.

At the lower level, the kernel communicates with the data *NoC* with *data_request* and *data_delivery* packets. The *pipe* and a message buffer enables packet retransmission to inter-task communication and inter-manager communication respectively.

The support for *I/O* communication uses a second *API*, with *IO_Receive*() and *IO_Send*() primitives, using a master/slave communication model. The *PE* is the communication master and the peripherals the communication slaves. At the lower level, the kernel communicates with the data *NoC* with *IO_request*, *IO_delivery*, and *IO_ack* packets. The *IO_Receive*() primitive uses the *IO_request* at the PE side and the *IO_delivery* at the peripheral side. The *IO_Send*() primitive uses *IO_delivery* at the PE side and the *IO_ack* at the peripheral side.

3.2 Control NoC - BrNoC

This Section presents the Control *NoC*, named *BrNoC* (broadcast *NoC*) The BrNoC is the first contribution of this Thesis, developed in cooperation with members of the research group. The design of this network aimed to create a network decoupled from the data network, allowing its use for fault tolerance - the theme of this Thesis [Fochi et al., 2018, Fochi et al., 2017], in security techniques [Caimi et al., 2017a, Caimi et al., 2017b, Caimi et al., 2018], system management (used throughout the protocols presented in this thesis). It is flexible and can be easily adapted for other applications.

The *BrNoC* [Wachter et al., 2017] is a dedicated NoC, decoupled from the data *NoC*. The BrNoC has the same topology of the data *NoC*, enabling to control each port individually (e.g. the North port in the dedicated *NoC* has an equivalent North port in the data *NoC*). The broadcast is the default transmission mode because it enables to reach PEs in case of disabled links, to notify several PEs with one message, and to transmit with low latency control messages.

In a broadcast, when a given port receives a message, it is processed and broadcasted to the neighbors routers (ports N, S, E, W), except to the port it came from. According to the transmission mode, the message may be transmitted to the port connected to the *NI* (local port). The broadcast acts as a wave traveling through the *NoC*. The BrNoC supports four transmission modes:

- ***brTgt*** (broadcast with a target): a specific *PE* is the target of this message. The message is broadcasted to all routers, but only the *PE* with the target address consumes it. This mode may be used to find a new path after a message discard adn to notify a specific *PE* to execute some action. The broadcast ensures that the message will be delivered even if a link/router is faulty or disable.

- ***brAll*** (broadcast to all PEs): all PEs consume the message. Therefore, all PEs are interrupted, and the message type defines the action the PE should execute. This mode may be used to freeze the tasks of a given application; send commands to PEs of same application ID; set wrappers to define a secure zone.

- ***brWt*** (broadcast without a target): all *BrNoC* routers consume the message without notifying the NIs. This mode executes actions related to the *BrNoC* management, as clearing specific data structures.

- ***unicast***: this message is an answer to a ***brTgt*** message. The ***unicast*** message follows the path defined by the ***brTgt*** message, in the reverse order to reach the source PE (backtrack process). This mode may be used to return a new path. Due to the limited payload size, each *BrNoC* router in the path sends a unicast message to the source router so that the fault-free path can be completely received.

Figure 3.6 presents the internal architecture of a *BrNoC* router, for a 2D-mesh topology. The router contains two control *FSMs* (Finite State Machines), two round-robin arbiters and a centralized *CAM* (Content Addressable Memory) memory. Routers have a small area footprint since they do not have input buffers (the *CAM* acts as a buffer shared by all input ports), and each flit encapsulates a single message.

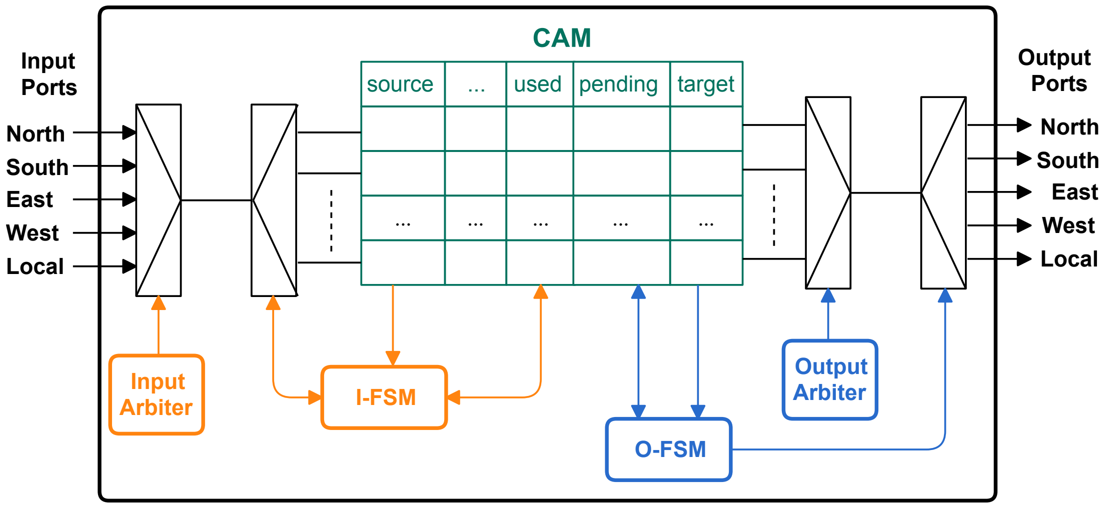


Figure 3.6 – *BrNoC* architecture.

The wrappers are connected to the control flow signals (*req*, *ack* in the control *NoC* - Figure 3.7). The control flow signals traverse the wrapper, if it is disabled. Considering the activation of the wrapper the *int_in_req* signal (internal value of the input request) is masked to 0 even if the *in_req* value is 1 in the external side of the *PE*. The value 1 in the wrapper value also set the *out_req* regardless of the *int_out_ack* value. This actions disable the message request from the neighbor *PE* and set the *out_ack* value to release it. From the neighbor *PE* the message was delivery. Equivalent behavior occurs when the request is generated internally (*int_out_req* signal).

Figure 3.8 details the flit structure (37 bits) and one *CAM* row (51 bits). Each *CAM* row stores the flit contents (to enable the broadcast) and control fields. The *flit* structure contains the fields: *message ID* (identification); *source address*; *target address*; message *type* (defines the action to execute and the transmission mode); message *payload*. The *tag*

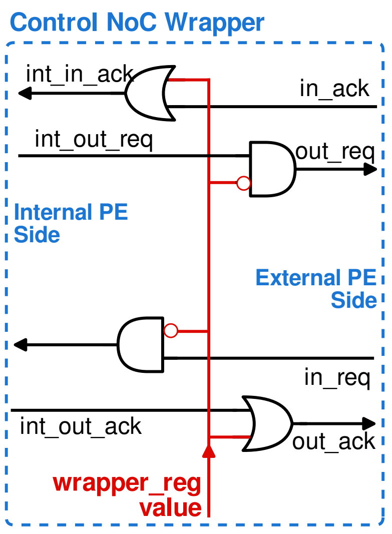


Figure 3.7 – Control NoC Wrapper logic.

to search in the *CAM* is the tuple *{msg ID, source address}*. Each brNoC link contains the flit structure plus the *req*, *ack* and *nack* signals.

The *CAM* size definition (number of rows) occurs at design time, and it is not a function of the system size, ensuring scalability. Smaller *CAM*s can increase the delay in handling the messages, while larger *CAM*s reduces avoid this delay at the cost of larger silicon area. The payload size may increase at design time to support services requiring larger data to transmit. The payload size is also a trade-off between the amount of data to transmit and the silicon area.

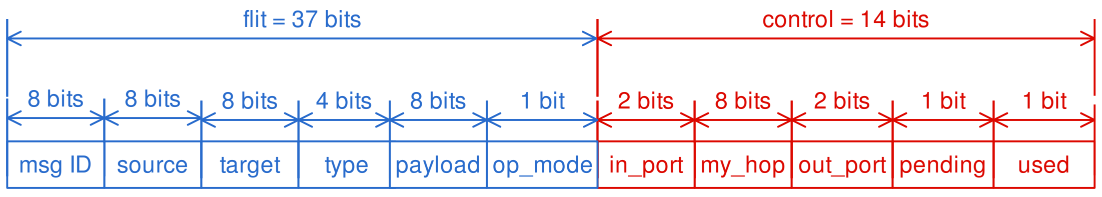


Figure 3.8 – Message (*flit*) and one row of *BrNoC CAM* memory.

The control structure of one *CAM* row contains the fields: *op_mode*, *in_port*, *my_hop*, *out_port*, *pending* and *used*. The *pending* field signalizes the presence of a message to be handled. The *used* indicates that the row is in use. The *in_port* stores the port identification from where the message comes from. The ***unicast*** mode uses the fields *my_hop* and *out_port*.

The control *NoC* has two operation modes (*op_mode* field): *global* and *restrict*. The *global* operation mode enables the control messages to pass through the wrappers, even if they are enabled. This operation mode enables PEs inside a secure zone to exchange messages with manager PEs. The *restrict* operation mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded. This mode enables a path discovery mechanism by the control *NoC*.

The I-FSM receives incoming messages and if necessary stores the message in a *CAM* row. A handshake protocol (*req*, *ack*, *nack*) controls the I-FSM. The I-FSM is initially in an idle state waiting for incoming messages (*req* asserted in a given port). The *input arbiter* chooses an input port to handle. Three conditions may assert the *ack* signal: (c_1) the *tag* is not in the *CAM*, and there is space in the *CAM*; (c_2) the *tag* is in the *CAM*; (c_3) failed or isolated port, where a wrapper force the *ack* signal. The assertion of the *nack* occurs when the *tag* is not in the *CAM*, and there is no space in the *CAM*. The router receiving the *nack* unsets the *req* and tries later (action discussed in the O-FSM). When condition (c_1) is satisfied the I-FSM executes the following actions:

- stores the message in a free position of the *CAM*;
- asserts the *pending* field to signalize that the message should be broadcasted;
- asserts the *used* field to signalize that the *CAM* row contains a valid message;
- stores the port identification selected by the arbiter in the *in_port* field (the size of *in_port* and *out_port* fields are a function of the number of router ports);
- in a search for a source-target path, the payload contains the distance from the current router address to the source address. This value is incremented and stored in the *my_hop* field.

Condition (c_2) ensures that requests to already visit routers are discarded, avoiding cyclic transmissions (i.e. deadlocks), and the end of the broadcast when all routers were visited.

The O-FSM handles the messages stored in the *CAM*, using the same handshake protocol. The *output arbiter* chooses a row to handle, according to the asserted pending fields. All broadcast modes propagate the message to the neighbors routers, except the *in_port*. According to the broadcast mode, the message also goes to all local ports (***brAll***), or to the local port that matches the router address with the target field (***brTgt***). The *pending* field is cleared when all broadcasted ports answer with an *ack*. If some broadcasted port answers with a *nack*, the arbiter selects another *CAM* row, enabling the selection of the current row again. An example of message type using ***brWt*** propagation is the CLEAR, responsible for freeing a *CAM* row, by clearing the *used* field. The ***unicast*** message uses the *in_port*, *my_hop* and *out_port* fields to answer a ***brTgt*** message. The ***unicast*** message forwards the message to the port defined in the *in_port* field.

Faults can compromise the path between a given source-target pair. The function of the *BrNoC* is to find an uninterrupted path between the communicating pair. Figure 3.9 presents an example of the procedure to find a new path. In this scenario, Router 1 communicates with Router 15 (XY path), but a fault in Router 11 interrupts the communication (the fault notification is sent to Router 11 neighbors). Using the *BrNoC*, Router 7 starts a

BROKEN_PATH message to Router 1. When the message reaches the Router 1, it starts a SEARCH_PATH message to find a fault-free path to Router 15 (Figure 3.9(a), red arrows).

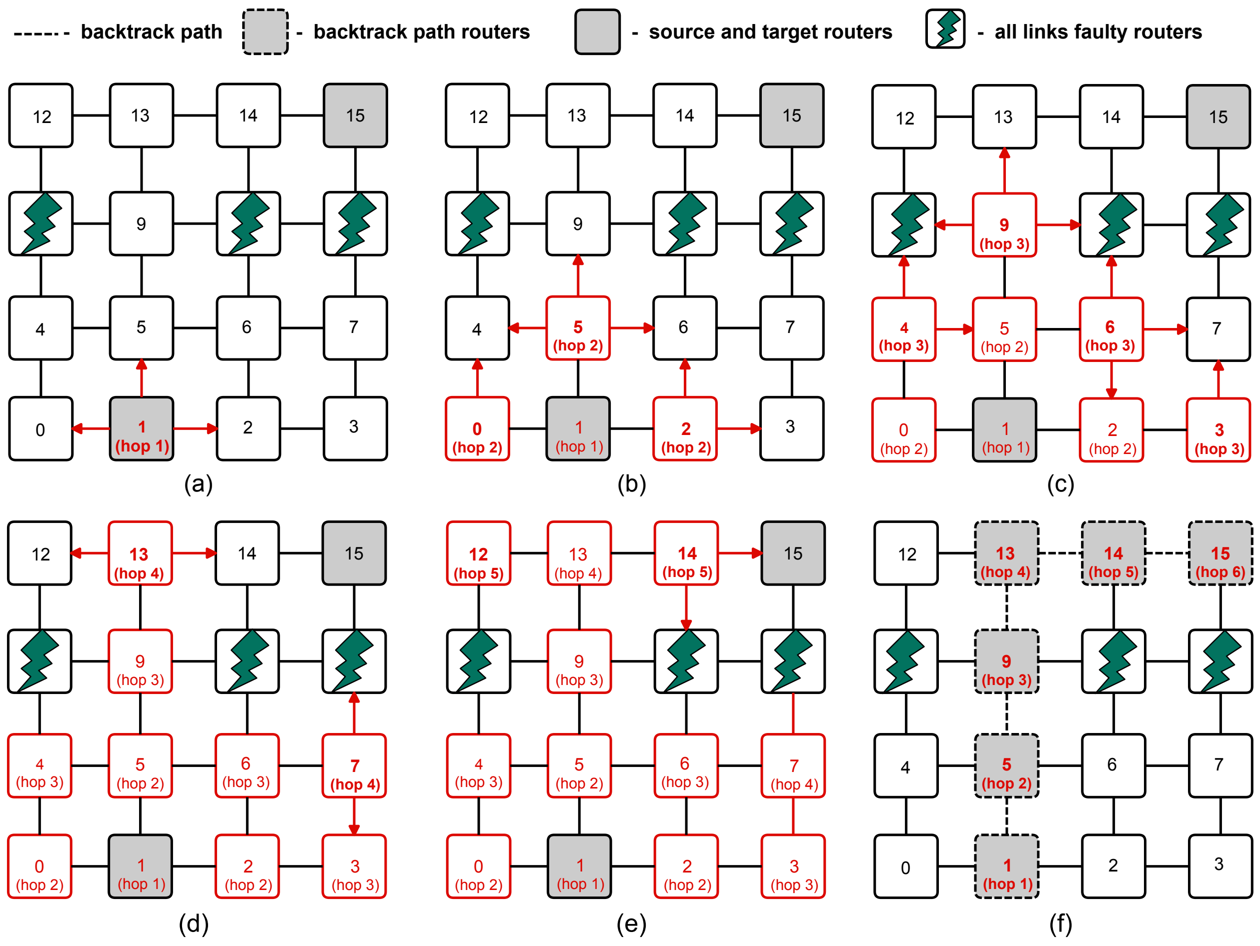


Figure 3.9 – Example of path discovery using the *BrNoC*.

Next, Routers 0, 2 and 5 receive the message through ports East, West and South, respectively. Then, these routers broadcast their messages to their neighbors (Figure 3.9(b)). In Figure 3.9(c), Routers 3, 4, 6 and 9 receive the message from ports East, South, West, and South, respectively and broadcast to their neighbors. As the Routers 8, 10 and 11 are faulty (wrappers are activated), they do not broadcast the messages (the wrapper force the *ack* signal to high). Note that the message sent by the Routers 3, 4, 6 and 9 is discarded in Routers 2 and 5 because these routers already have received the message from the same source (*msg ID*/*source address* stored in *CAM*). Next, Figure 3.9(d), the message is received in ports West and South of Routers 7 and 13, respectively, and sent to their neighbors. In the next hop (Figure 3.9(e)), the message is received in Routers 12 and 14, and finally, in Figure 3.9(f), the SEARCH_PATH message reaches the target, starting the answer step with BACKTRACK messages.

In the answer step, each router in the path sends a BACKTRACK message to the source router. Initially, Router 15 sends a BACKTRACK message to Router 1 through the West port (information stored in the *in_port* field). Next, Router 14 propagates the first message, and then transmits a new BACKTRACK message to Router 1 with the payload having the contents of the *out_port* field. Each router in the path repeats this process, propagating the previous BACKTRACK messages and sending a new one. The *my_hop* field controls the process, finishing when the source router receives all BACKTRACK messages (*my_hop*=1). Therefore, the source router receives a number of BACKTRACK messages equal to the number of hops in the path to the target. Each one of these messages contains in the payload the port to reach the destination router.

When Router 1 receives the BACKTRACK message, the *PE* is interrupted to compute the source routing path to Router 15. Then, Router 1 resends the lost message and all subsequent packets to this destination using the source routing path, which is stored in an *OS* structure. The process to find a new path to a given target is executed once, only when the fault is detected.

Table 3.1 shows examples of messages types with the respective transmission mode, operation mode and purpose in the context of this Thesis. The message type operation will be explained in Chapters Chapter 5 and Chapter 6.

Table 3.1 – Examples of messages types and purposes of the *brNoC*.

Purpose	**Message Type**	**Transmission Mode**	**Operation Mode**
Path Discovery	BROKEN_PATH	***brTgt***	Global
	SEARCH_PATH	***brTgt***	Restrict
	BACKTRACK	***unicast***	Restrict
Recovery	FAIL_CPU	***brAll***	Global
	FAIL_INJ_LINK	***brAll***	Global
	FREEZE	***brAll***	Global
	UNFREEZE	***brAll***	Global
	BROKEN_RECEPTION	***brAll***	Global
	WAIT_KERNEL	***brTgt***	Global
	SEND_KERNEL	***brTgt***	Global
	VGM_READY	***brAll***	Global
System Management	START_APP	***brAll***	Global
	END_TASK	***brTgt***	Global

Figure 3.10 presents an example of the procedure to notify a fault in the manager (`FAIL_CPU_message`) and a `FREEZE_message` to hold a cluster using the control *NoC*. In this scenario Figure 3.10.a, Manager 0 became faulty. A message to notify the Manager Pair PE2 is sent in broadcast. Using the control *NoC*, Router 0 starts the `FAIL_CPU_message` to Router 2 (Figure 3.10.a, red arrows). Next, Router 4 and 1 receive the message through ports South and West respectively. Then, these routers broadcast the received message to their neighbors (Figure 3.10.b). After 3 hops (42 clock cycles), Router 2 receives the message (Figure 3.10.c). Finally the fault notification is sent to the CPU.

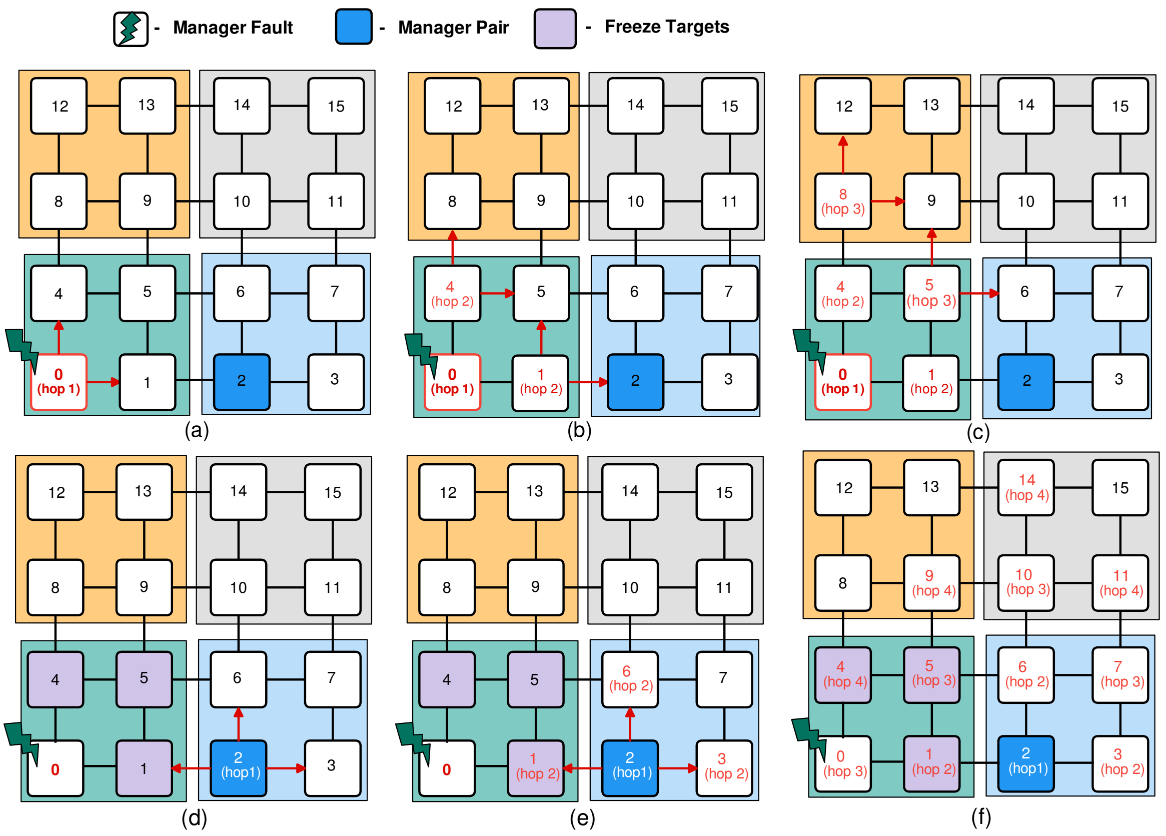


Figure 3.10 – Example of fault and freeze notification using the *BrNoC*.

Figure 3.10.d presents an example of a `FREEZE_message`. In this scenario (Figure 3.10.d, red arrows) Router 2 starts the broadcast of the `FREEZE_message` and with target Routers 1,4 and 5. Next, Routers 1, 3 and 6 receive the message through ports East, West and South respectively. Then, these routers broadcast the received message to their neighbors, Figure 3.10.e. Finally the `FREEZE_message` notification reaches the targets. In this case the broadcast spends 4 hops (56 clock cycles) to achieve all targets.

One may ask: "is it possible to implement the methods proposed in this Thesis without the *brNoC*?". The benefits of using the *brNoC* include : (*i*) the broadcast transmission enables to reach a large number of targets quickly, without the need to know the PE addresses to freeze/unfreeze; (*ii*) the control messages may reach their targets even in the presence of faults in the NoCs; (*iii*) the control traffic is isolated from the applications' traffic, not interfering in their performance. Thus, it would be possible to use only the Data *NoC*, but the cost in terms of the protocols' performance would be high (unicast messages instead of broadcast ones) and interference between control and data packets.

3.3 Application Injector

This Section presents the Application Injector module, which is an external peripheral. The Application Injector module is the second contribution of this work, developed by the Author of this Thesis. The support to connect external devices in the *MCSoC*, and the application Injector module are part of the baseline architecture, which evolved from *HeMPS* to *MEMPHIS* [Ruaro et al., 2019].

Support for peripherals required simple modifications at both hardware and software levels. At the software level, it was necessary to implement the peripheral communication API, described in Section 3.1.2. At the hardware level, only the "switch control" module (router) was modified to differentiate data packets from peripheral packets. When a peripheral packet reaches the destination router, it is transferred to the output port, to which the peripheral is connected.

In the original baseline architecture model (*HeMPS*), the Global Manager (*GM*) was mapped statically (address$_{0,0}$) and connected to a repository (external memory with the applications' codes). The *GM* was responsible for sending applications to the system, with one connection with the repository. This original *MCSoC* model presents problems related to both, a general purpose architecture and fault tolerance:

- In a general purpose architecture, the *MCSoC* system is connected to external interfaces, such as network interfaces (e.g., Ethernet) or other communication interfaces. The baseline architecture needs to be modified to reflect the structure of real systems.

- In a general purpose architecture, the *MCSoC* system can be connected to shared external memories and hardware accelerators.

- As for fault tolerance, the existence of only one PE connected to the repository represents a single point of failure.

These three issues led to the development of the architecture with the possibility of connections in the external boundaries of the *MCSoC* and, specifically in the context of this Thesis, to the development of the Injector module that approximates the *MCSoC* to real systems and eliminates the single point of failure if the Injector is replicated.

Figure 3.11 presents the Injector structure. This module communicates with the two NoCs. The *brNoC* is used for exchanging control messages, while the data *NoC* is used for sending messages with longer payloads, such as applications' descriptors and object codes. The injector has five state machines controlled by the Application Injector Machine (*AIM*). The injector emulates a Data router (Data *NoC* in / Data *NoC* out) and Control router (BrNoC in / BrNoC out) behavior. In this way, any hardware module can insert applications, as long as it respects the communication protocol for applications' admission. The communication with the injector is transparent for the MCSoC. No special packet is needed for the exchange of messages.

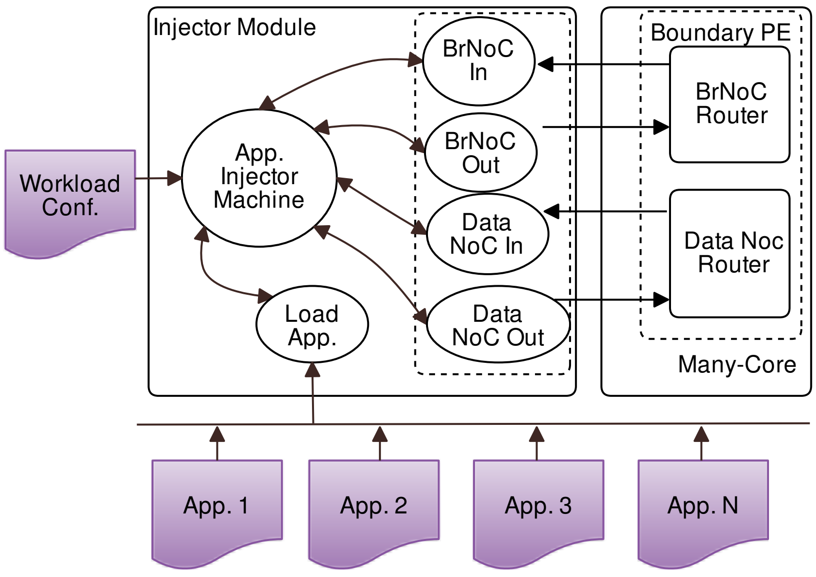


Figure 3.11 – Application Injector connection with the MCSoC.

At design time occurs the following actions:

- the kernel of each manager receives a tuple {Injector ID, address, port}, enabling the communication between managers and injectors/peripherals;
- each injector/peripheral receives a unique ID, enabling it to receive/send packets from/to the many-core;
- a "workload configuration" file is created with the tuples {Application ID, injection time, [optional] static mapping};
- the object code of each application is created and stored in a file.

When the simulation initiates, the *AIM FSM* starts reading the workload configuration file. For each application (*app*) to be injected into the system, the protocol described later in Section 6.2 exchanges messages through the *brNoC* with the *VGM* in such a way to receive the cluster ID to send *app*. The injector allows the injection of one application at a time. If the workload configuration file specifies two applications with the same injection time, they are inserted into the system sequentially. FSMs involved in this process: *AIM*, BrNoC in, BrNoC out.

Next, the *AIM FSM* reads the *app* descriptor, transmitting it to the selected cluster ID. FSMs involved in this process: *AIM*, Load App, Data NoC Out.

The selected cluster maps the *app* tasks and requests the object codes. FSMs involved in this process: Data NoC In, *AIM*.

Finally, the *AIM FSM* reads the *app* object codes and transmits them to the system. FSMs involved in this process: *AIM*, Load App, Data *NoC* Out.

The injector module is described in SystemC. Chapter A shows the *FSM* diagrams.

4. SYSTEM MANAGEMENT RECOVERY OVERVIEW AND FAULT MODEL

This Chapter presents in Section 4.1 a general view of the system management recovery method for the *VGM* or *LM*. This presentation is required to provide a holistic perspective of the Thesis approach before detailing the methods on the next Chapters. Section 4.2 presents possible situations handled by the method and a high-level flow chart, with the actions executed at each scenario. Section 4.3 presents the fault model, and the fault-tolerant techniques available in the literature organized according to their proposals.

4.1 Proposed Recovery Method Overview

This Section presents a management recovery method overview. The recovery method is similar for the *VGM* and LMs. However, each one is explained separately because the *VGM* communicates with the Injectors. The fault detection mechanism (out of the scope of the present work) detects a permanent fault in the processor of the *VGM*. The fault detection fires the following protocol:

1. The fault detection signal, in the *VGM*, induces the *brNoC* to send three control messages:

   (a) broadcast message targeting all SPs managed by the faulty manager processor, VGM_F, to freeze all tasks managed by VGM_F (note that due to the reclustering process some tasks may be executing in another clusters).

   (b) broadcast message targeting the Injectors to stop the admission of new applications, and eventually, interrupts the admission of a new application if this application should execute in the cluster managed by VGM_F.

   (c) broadcast message targeting the *Manager Pair* (manager processor supervising the *VGM*) to start the kernel migration process.

2. The *Manager Pair* evaluates if there is in the cluster managed by VGM_F a free *SP* (*SP* in an idle state, not executing tasks). If this condition is not satisfied, the *Manager Pair* selects an *SP*, migrating the running tasks to available SPs to neighbor clusters. The result of this step is the selection of an *SP* to receive the code and data of VGM_F, named $SP_{candidate}$.

3. Migrate the memory contents of VGM_F to $SP_{candidate}$.

4. After migrating the VGM_F to $SP_{candidate}$, the *VGM* restarts in this processor, unfreezing the tasks managed by it.
5. If an application admission was interrupted, the *VGM* releases the reserved SPs.
6. The new *VGM* restarts the communication with the Injectors. If an application admission was interrupted, the Injector restarts the protocol.

The Manager Recovery Method is similar for the *LM*. The main difference is that the Injectors are free to inject new applications into the system. Only the cluster belonging to the LM_F freezes its execution and cannot receive new applications. After the recovery process, the injector starts the application injection protocol again. The *VGM* does not choose the cluster while the *LM* is faulty. The cluster can receive new applications when the new *LM* restarts.

4.2 Actions Executed by the Recovery Protocol

Figure 4.1 exemplifies two possible situations handled by the proposed recovery method, using as example two 4x2 many-core instances, with two 2x2 clusters. In Figure 4.1(a), $LM_{2,0}$ is faulty and $SP_{3,0}$ is free, i.e., there is no tasks executing on it. In this case, the proposed recovery method migrates the kernel from $LM_{2,0}$ to $SP_{3,0}$. In Figure 4.1(b) all SPs of the cluster managed by the faulty *LM* execute tasks. In this scenario, the recovery method migrates tasks executing in the cluster to another cluster before migrating the kernel.

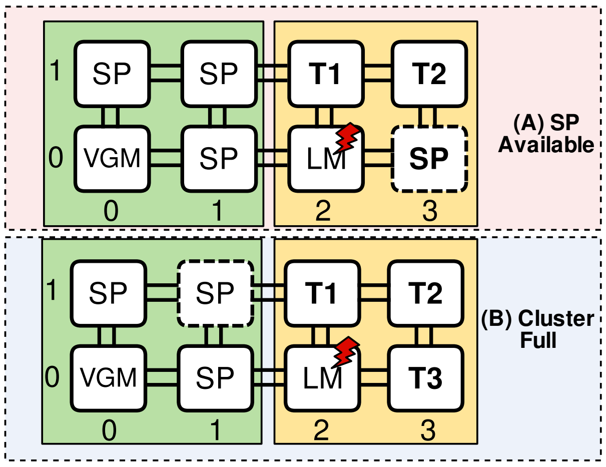


Figure 4.1 – Scenarios handled by the recovery method: (a) cluster with available SPs; (b) cluster with all SPs executing tasks.

The proposal starts by defining *Manager pairs*. Each Manager selects its pair at runtime. A pair of Managers are responsible for supervising each other, by exchanging periodically control messages, or for receiving a fault message.

Figure 4.2 presents the actions taken when a Manager presents a permanent fault.

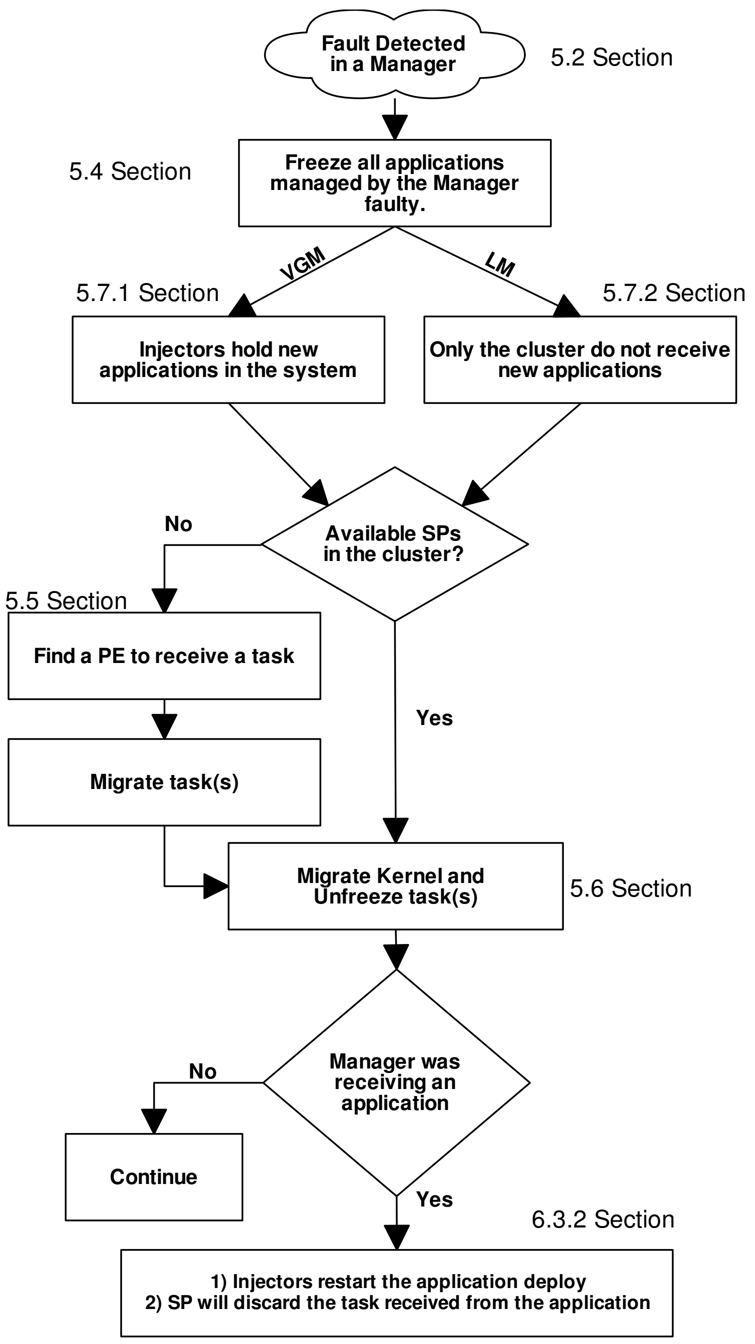

Fault Detected in a Manager
5.2 Section
5.4 Section
Freeze all applications managed by the Manager faulty.
VGM
LM
5.7.1 Section
5.7.2 Section
Injectors hold new applications in the system
Only the cluster do not receive new applications
No
Available SPs in the cluster?
5.5 Section
Find a PE to receive a task
Yes
Migrate task(s)
Migrate Kernel and Unfreeze task(s)
5.6 Section
No
Manager was receiving an application
Continue
Yes
6.3.2 Section
1) Injectors restart the application deploy
2) SP will discard the task received from the application

Figure 4.2 – High-level flow chart, with the actions executed by the recovery protocol. Above the rectangles, it is inserted the section detailing the procedures.

When a permanent fault is detected in a Manager (faulty Manager, or M_F), its Manager pair (healthy Manager, or M_H) is notified by a broadcast control message. The M_H starts the recovery method. The M_H immediately inject a *freeze* message to all the PEs. All tasks managed by M_F stop their execution. Next, M_H evaluate the PE location to receive the functions executed by M_F. If there is an available *SP* in the cluster, i.e., with no tasks assigned it, the kernel migration process starts. Otherwise, it is necessary to release an *SP* of the cluster managed by M_F to another cluster. This action is done by migrating one or more tasks to a free *SP*. When the task migration finishes, the kernel migration begins. After the kernel migration, the *PE* that received the kernel assumes the role of the previous M_F.

The recovery process isolates the faulty processor, resulting in a graceful degradation of total processing power. However, the system continues to operate even in the presence of permanent faults in the manager processors.

4.3 Fault Model

The focus of this Thesis is not the fault detection, but the recovery method for a fault recovery in a Manager PE. This work assumes:

- Healthy modules of the *PE*: memory, *DMNI*, data and control NoCs. A usual method to protect the memory is the usage of *ECC* (Error Correction Codes). The *DMNI* is a small hardware module with two state machines and a buffer. This module may be protected by hardware replication and adoption of *ECC* in the buffer. Besides the NoCs be considered healthy, it is possible to detect transient faults [Fochi et al., 2015], and according to the severity of the transient faults trigger the proposed protocol.
- *PE* module subject to faults: *CPU*. The proposed method is fired when a permanent fault is detected in a manager *PE*. The basis of the fault recovery method is a monitoring process between manager PEs. When a fault is detected on a manager *PE*, a message is sent to its Manager Pair. When a manager *PE* has a permanent fault, the recovery process starts. The goal of the recovery process is to transfer the memory contents of the faulty manager to a healthy *PE*. The *DMNI* of the faulty manager *PE* handles this process.

4.3.1 Fault Detection Mechanisms

This section presents examples of fault detection methods, that can be applied to the *PE* modules, and used by be current work. All techniques cited bellow can initiate the proposed protocol. A rich literature with methods to test the PE modules is available, with approaches adopted at different levels or modules.

– Fault detection and management at the system level. The Madness project [Meloni et al., 2012] adopts two approaches to detect faulty processors: self-testing using a pre-computed signature for non-critical applications, or N-modular redundancy at the software level. Using these methods, the Authors present a system level adaptive and fault-tolerant techniques to reduce the performance loss by using dynamic remapping (task migration) of faulty PEs. However, it is not defined if the *PE* is a system manager. It can be inferred that it is an *SP* because it executes tasks. In [Boraten and Kodi, 2016] the Authors propose a Runtime Module Configuration with a 3-mode configurable encoder. The goal is to change the encoder mode according to the number of faults occurring at the *NoC* links. The method encodes packets and optimizes the fault coverage of the *NoC*. [Kamran et al., 2016] focus on a test framework for a specific cluster, applying the technique to PEs that are in idle mode, deactivating the *PE* if a fault is detected. However, this work does not define who is the manager in the *MPSoC* or in the cluster, nor what happens if the manager fails. [Bolchini et al., 2013] focus on system management and fault-tolerance on PEs. This work uses software-based techniques to detect faulty PEs. A macro technique analyzes the failures to modify the system behavior. An element called fabric controller dispatches the applications to the various tiles. Fault-tolerance is applied only to slave tiles. [Paul et al., 2015] propose a management approach that focuses on resource sharing adaptively, but it does not focus on fault-tolerance. [Martins et al., 2016] focus on power management using techniques such as *DVFS*, clock gating, and task migration. [Domingues et al., 2018] propose fault-tolerance in the manager cores applied only to the communication protocol of a publisher-subscribers approach. [Tsoutsouras et al., 2017] propose a fault tolerance in the manager cores, with the *MPSoC* hierarchically organized in clusters.

– Fault detection at the processor level. In [Braak et al., 2010] a general-purpose device (*GPD*) creates a test pattern, sending it to the processor. The processor applies the test pattern and sends the results back to the *GPD*. Faulty tiles are bypassed and replaced by another processor via an embedded resource manager implemented in software. The Authors in [Walters et al., 2011] adopt a software-based fault tolerance, process-level replication, thread-level replication, kernel level checkpoint/rollback and distributed heartbeat implementation.

– Fault detection at the router level. The proposal in [Fick et al., 2009b] inserts multiplexers at the input ports to enable port swapping, and a bypass bus enables to connect input ports to output ports when the internal crossbar fails. In [Zhang et al., 2012] present a dual-input crossbar design targeting performance and power reduction. The crossbar duplication enables fault tolerance at the router level. When a crossbar failure is detected, all the inputs ports are forwarded to another crossbar. The proposal in [Yu et al., 2011] focuses on transient faults in the router using *ECC* to prevent packet loss, incorrect routing, and network congestion. An fault detection module request re-computation if a fault is detected. It also includes an fault correction module after the crossbar to prevent fault propagation. [Chen et al., 2017] propose a fault-tolerant routing algorithm for NoC focused on path discovery to avoid congestion of *NoC* in the presence of faults.

– Fault detection at the link level. In [Vitkovskiy et al., 2012] the Authors propose a fault-tolerant method with a gracefully degrading link-level, proportional to the number of faults detected in the link. In [Veiga and Zeferino, 2010] the Authors implement an fault recovery technique for NoCs with the goal to protect network links against crosstalk effects using Cyclic Redundancy Check (*CRC*) modules. [Bhowmik et al., 2016] also focus on *NoC* only to detect link failures between routers using a technique of sending test packets between routers.

5. SYSTEM MANAGEMENT RECOVERY METHOD

This Chapter presents the recovery method of the system management functions, action required when a Manager PE (*MP*) becomes faulty. The fault in MPs is assumed permanent, and the fault detection mechanism is out-of-the-scope of the present Thesis. The method handles faults, at runtime, in both MPs, Virtual Global Manager (*VGM*) and Local Managers (*LM*). The recovery method migrates the management functions of the faulty MP to a healthy PE. Briefly, after a fault notification, the method selects a Slave Processor (*SP*) to become the new MP, freezes the tasks managed by the faulty *MP*, migrates the memory contents to the new MP (*kernel migration*), restarts the processor, and unfreezes the tasks without restarting them. This recovery method is an original contribution of this Thesis, and a relevant feature of the proposed method is to preserve the management context without saving it in redundant structures.

Section 5.1 to Section 5.6 presents the mechanism to deploy the recovery method when an *MP* presents a fault. Section 5.7 uses the mechanisms presented along this Chapter to show how the many-core management can be recovered in the presence of faults, either in local or global managers. Section 5.8 concludes this Chapter, summarizing its contributions.

5.1 Manager Pairs Definition

The recovery method starts by defining *ward pairs*. Each *MP* (*LM* or *VGM*) selects its pair, and they exchange messages, named `ward_messages`, to determine if they are alive. At system startup, the *ward pairs* are physically aligned, but after migrating an MP to a new position this organization changes. Thus, the communication between MPs to manage the *ward pairs* occurs through the *brNoc*, enabling the communication using the MP identifier and not its physical address.

The method defines at system startup *ward pairs*. Consider Figure 5.1 as an example. The first step of the method defines horizontal *ward pairs*. In the Figure they are: {$VGM_{0,0}$, $LM_{3,0}$}, {$LM_{0,3}$, $LM_{3,3}$} and {$LM_{0,6}$, $LM_{3,6}$}. If the number of *LM* columns is odd, the second step of the method defines vertical *ward pairs* at the rightmost *LM* coordinate. In this example, one vertical *ward pair* is created, {$LM_{6,6}$, $LM_{6,3}$}. Finally, as $LM_{6,0}$ has no ward pair to supervise, its ward pair is the last *LM* address, $LM_{6,3}$. Thus, $LM_{6,3}$ is in charge to monitor the status of $LM_{6,0}$ and $LM_{6,6}$.

$$\begin{tikzcd}
LM_{0,6} \arrow[r, leftrightarrow] & LM_{3,6} & LM_{6,6} \arrow[d, leftrightarrow] \\
LM_{0,3} \arrow[r, leftrightarrow] & LM_{3,3} & LM_{6,3} \\
VGM_{0,0} \arrow[r, leftrightarrow] & LM_{3,0} & LM_{6,0} \arrow[u]
\end{tikzcd}$$

Figure 5.1 – Example of ward pairs definition.

Each MP runs a supervision function for triggering the recovery process when it detects the fault on its pair. Note, in the Figure, that $LM_{6,0}$ sends periodically `ward_messages` to $LM_{6,3}$, and its supervision function is disabled. Its supervision function is disabled because the MP responsible to recover $LM_{6,3}$ is $LM_{6,6}$.

5.2 Fault Detection Notification

Figure 5.2 presents the first method to trigger kernel migration. Event 1 in Figure 5.2 corresponds to `ward_messages` exchanged periodically between MPs (LM0 and LM1), through the *brNoC*. The advantage of the method is that these messages do not interfere with the traffic in the Data *NoC*. The interval definition between `ward_messages` is a design-time parameter. Large periods delays the time to recover from a faulty *MP*, while short periods may lead to false positives. False positives may occur if the MP is executing management functions, delaying the answer to its pair, which will consider it faulty, starting the recovering process.

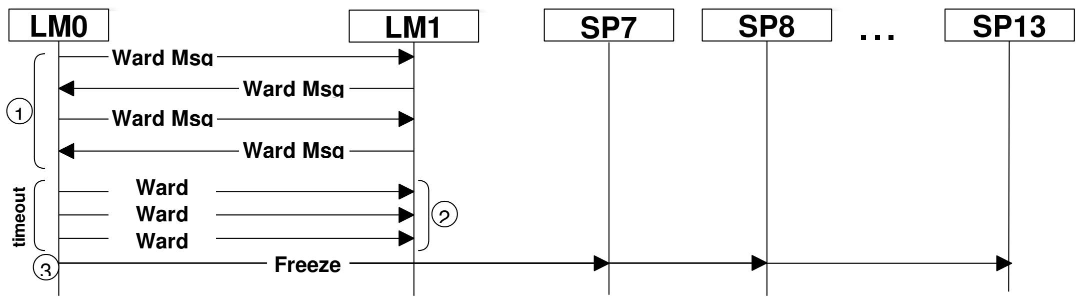


Figure 5.2 – Protocol to detect a faulty *MP* using `ward_messages`.

We adopt experimentally an interval of 1 ms, as it represents a good trade-off between recovery time and execution time overhead in MPs. MPs transmit `ward_messages` through the *brNoC*, using the MP identifier and not its physical address. As *brNoC* uses broadcast to transmit its messages, the `ward_messages` reaches the destination MP even is there is a fault in the *NoC*.

If an *MP* sends three ward_messages without receiving any answer from its pair (event 2), the *MP* that did not reply is considered faulty, starting the recovery process (event 3). Although effective, this method proved to be slow to detect faulty MPs, because it requires three unanswered messages, in addition to the fact that the interval between messages needs to be evaluated to avoid false positives.

A second method has been developed to reduce the fault detection time. This method includes a new hardware module, name *Fail Wrapper Module* (*FWM*), responsible for: (*i*) isolate the control signals of the MP when the fault detection mechanism detects the fault; (*ii*) generate a fault message for its *ward*. The main advantage of this method is the isolation of the faulty *MP* by means of wrappers as soon as the fault is detected. This prevents the MP memory contents from being modified, and the occurrence of Byzantine faults due to the fault on this component.

When a fault notification occurs, the *FWM* isolates the faulty *MP*, and generates a message, fail_CPU_message to the *ward MP* – Figure 5.3(1). Upon the reception of the fail_CPU_message, the *ward* MP start the recovery process. This method replaced the first one, ward_messages.

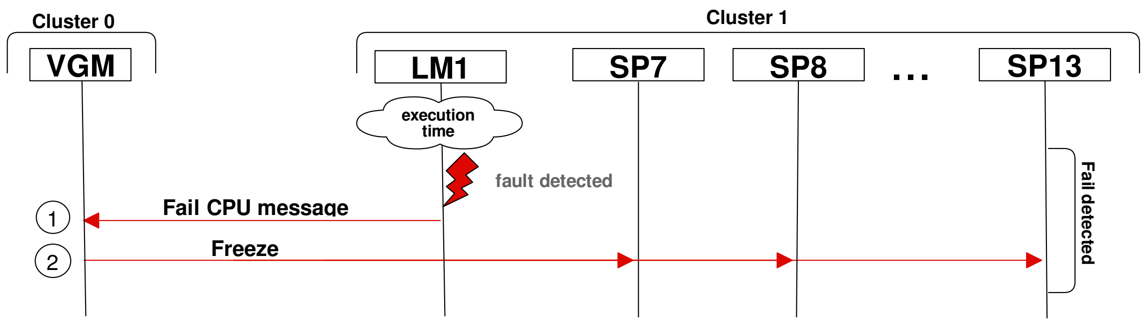


Figure 5.3 – Fault notification using a fail_CPU_message.

Figure 5.4 presents the sequence of events handled by the *FWM*:

1. *MP* becames faulty;
2. the fault detection module notifies the *FWM* that a permanent fault was detected;
3. the *FWM* isolates the *MP*, disabling its access to any hardware module, including its local memory;
4. the *FWM* inject in the *brNoC* router a fail_CPU_message.

Note that the fault detection module can also work with transient faults that occur frequently, which point out that the *MP* is suffering from aging effects. Given a threshold on the number of transient faults for a given period, this module may decide that the MP is

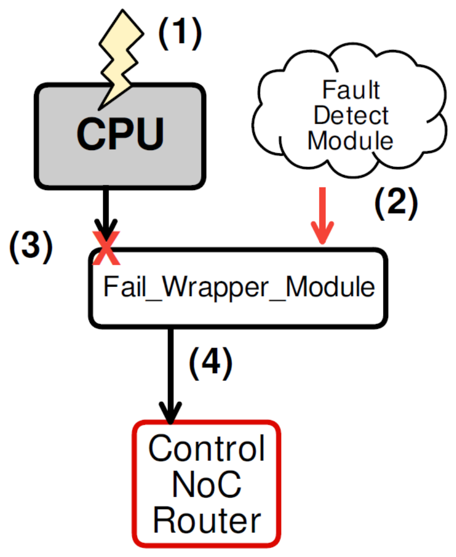


Figure 5.4 – Fail Wrapper Module (*FWM*).

faulty. The process of detecting aging effects is not in the scope of this work but it is a feature to consider in the development of the fault detection module.

5.3 Manager Candidate Definition

Figure 5.5 presents the Manager Candidate selection process. Figure 5.5(a) presents the many-core configuration at the startup. The Figure presents a 6x6 many-core, organized in four clusters, with four MPs. The closest *SP* to its *MP* is the manager candidate – $SP_{candidate}$.

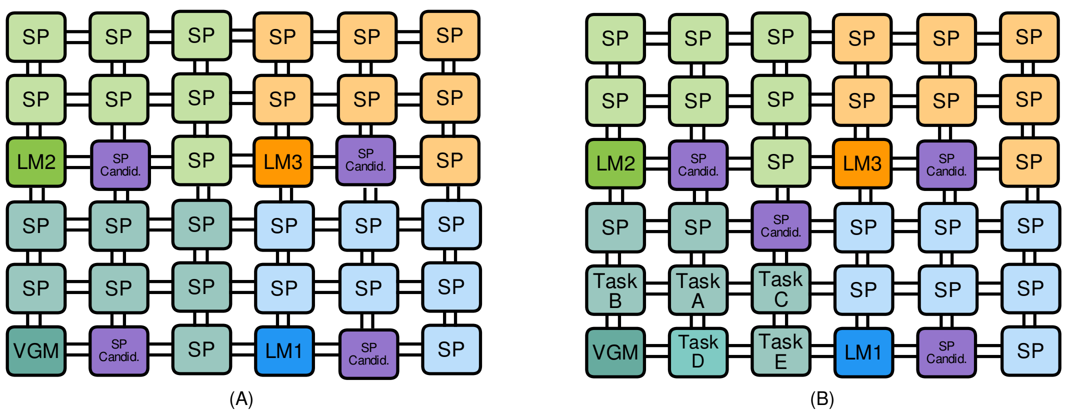


Figure 5.5 – Manager Candidate ($SP_{candidate}$) selection - (a) startup; (b) selection after a new application admission.

The admission of a new application into the many-core is a two-step process. First, the *VGM* selects a cluster to receive the new application according to some criteria, as the number of available SPs. Next, the *LM* (or *VGM* if the selected cluster is the one managed by the *VGM*) maps the tasks into the cluster. After the mapping procedure, the *LM* or *VGM*, verifies if the $SP_{candidate}$ received a task. In this case, the rule to select a new $SP_{candidate}$ is to choose the *SP* with the minimum number of tasks assigned to it. Thus, after an application mapping, if the $SP_{candidate}$ address changed, the *MP* transmits it to its *ward*: (*i*) new address; (*ii*) the number of tasks executing in the $SP_{candidate}$ because if it is different from zero, the *ward* will manage the migration of the tasks executing in the $SP_{candidate}$.

Figure 5.5(b) presents a manager candidate selection scenario. The *VGM* mapped a new application in its cluster, mapping task D in the $SP_{candidate}$. The *VGM* searches a new *SP* with no tasks assigned to it, and chooses a free *SP* as the new $SP_{candidate}$. After the selection process, the *VGM* sends a `Master_Candidate_Message`, using the *brNoC*, to LM1 (its *ward*), updating the $SP_{candidate}$ address.

5.4 Freeze & Unfreeze Messages

When a *MP* fails, the tasks it manages should be suspended to prevent control messages from being lost. This suspension process is called *freezing*. Since the *MP* is faulty, its *ward* executes this action.

Freeze and unfreeze are control actions to stop or release the execution of a set of tasks. Figure 5.6 presents the freezing process that starts with a healthy MP (LM1) transmitting in broadcast a `freeze_message`, by the *brNoC*, having in its payload the address of the faulty manager (*MF*). Any *SP* receiving a freeze message verifies if it has tasks managed by *MF*. In this the case, all tasks of this *SP* are freezed. Otherwise, the message is discarded. The broadcast transmission of the freeze message enables to stop tasks in *SP*s managed by *MF* executing in other clusters, due to the reclustering process. The freeze message does not stop the tasks immediately. To avoid messages losses, the task must be in a safe state. A safe state is defined as: the task to freeze should be ready to be scheduled by the kernel, and there is no pending request for messages. For example, if a task is in a waiting state, this means that the task requested a message to a producer task. Thus, the producer receives the request and at some moment inject messages into the *NoC*. Such procedure ensures that when a given task stops, there are no messages generated by the task in the data *NoC*. Thus, all tasks managed by *MF* goes to the freeze state, avoiding their scheduling by the kernel.

After the recovery process, the new *MP* sends an `unfreeze_message`, also in broadcast. This message unfreezes the tasks managed by the new *MP* and also transmits the new *MP* address to the *SP*s of the cluster.

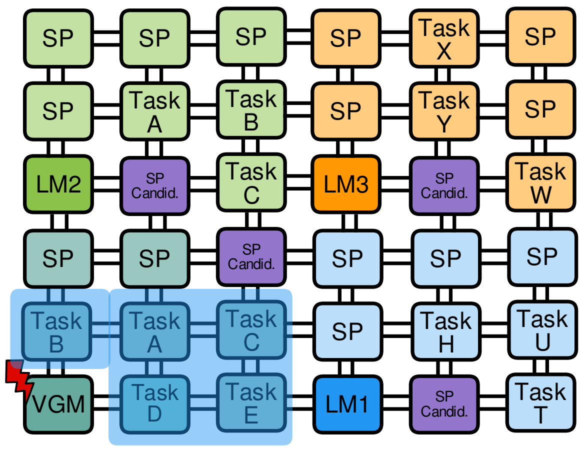


Figure 5.6 – Freeze process on the cluster managed by the *VGM*. LM1 injects the `freeze_message`.

5.5 Task Migration

Tasks migration is an action required to release an *SP* to receive the manager processor software if this *SP* is executing one or more tasks.

Figure 5.7 presents a possible scenario handled by the recovery protocol with task migration. In this scenario, it was detected a permanent fault at the *LM*. All SPs of this cluster have at least one task in execution, being $SP_{2,1}$ the $SP_{candidate}$. In Figure 5.7(a) occurs the fault detection in the *LM*. The fail wrapper module (*FWM*) notifies the fault by injecting a *fail_CPU_message* in the *brNoC*. The *VGM* knows that the $SP_{candidate}$ is executing task C (Section 5.3). Thus, it is necessary a task migration before the recovery process. In Figure 5.7(b) task C migrates from $SP_{2,1}$ to $SP_{1,1}$, in another cluster. When the task migration finishes, the kernel migrates to $SP_{2,1}$ (Figure 5.7(c)).

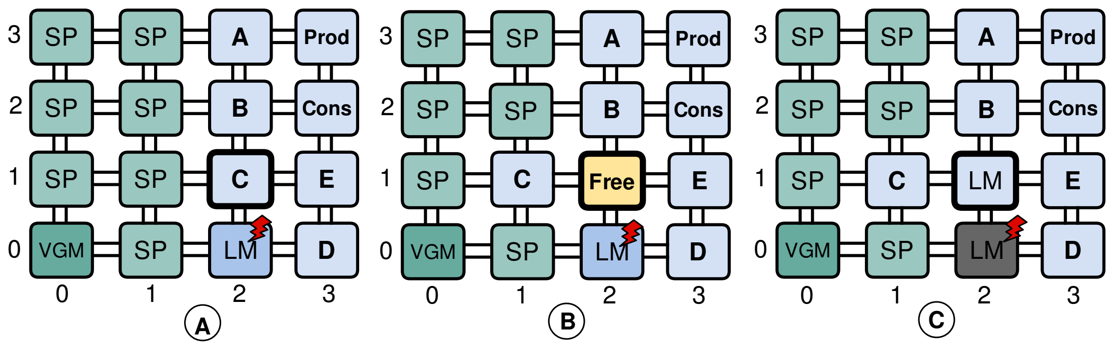


Figure 5.7 – Task migration to release an *SP*. (A) Fault detect at *LM*; (B) task C migrated from $PE_{2,1}$, to $PE_{1,1}$; (C) *LM* migration from address (2,0) to (2,1).

Figure 5.8 presents the recovery method protocol. After the application mapping phase (events 1,2 in the Figure), at a given moment a permanent fault is detected at *LM*1 (3), and its *ward* is notified. The *ward* (*VGM*) freezes the tasks managed by *LM*1 (4), verifying that a task migration is required. The *VGM* selects SP1 to receive the tasks executing on SP7. The *VGM* sends a migrate task message to SP7 (5). SP7 sends to SP1 a set of messages with the task contents. After receiving all messages related to the task migration, SP1 notifies to all application's tasks the new location of the migrated task (6). The task migration ends with SP1 notifying *VGM* the end of the migration process (7).

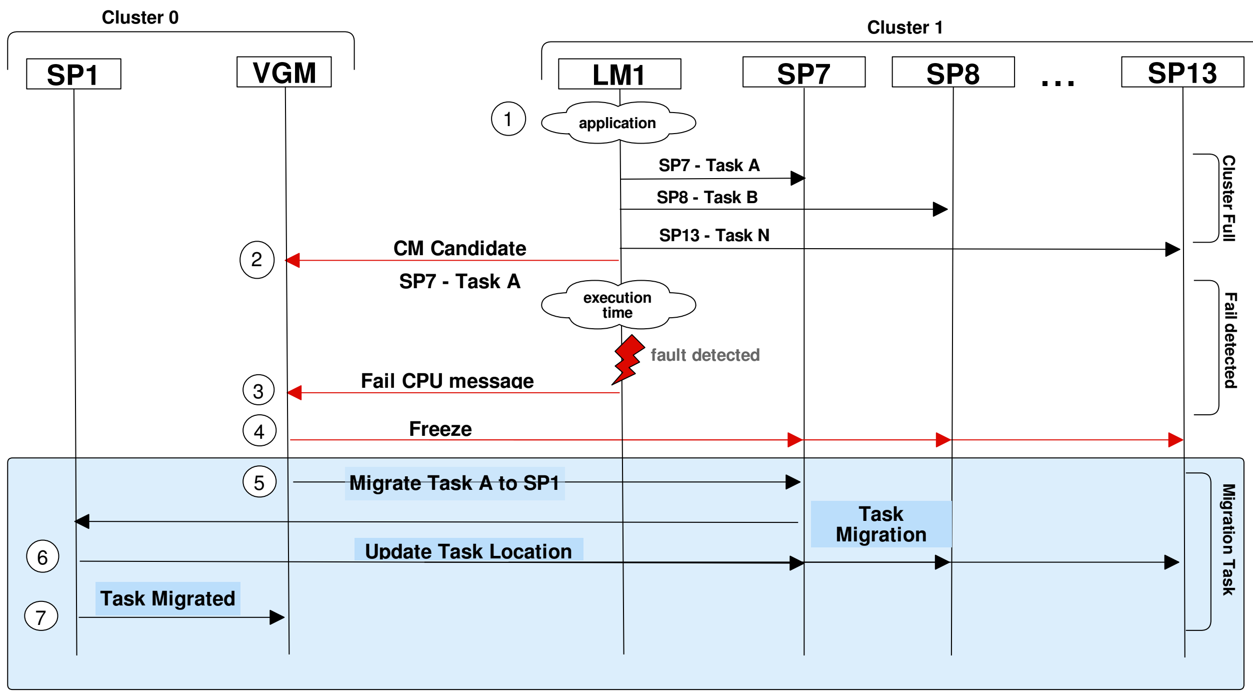


Figure 5.8 – Sequence diagram of the recovery protocol steps with task migration. Black arrows: messages transmitted through the Data *NoC*. Red arrow: messages transmitted through the Control *NoC*.

The selection of the *SP* that will receive the task from the cluster with a faulty *MP* occurs in the cluster of the healthy MP (*ward* of the faulty *MP*). It is assumed the presence of a free *SP* in the cluster managed by the *ward MP*. There are methods available in the *MP* to circumvent scenarios where this cluster has no available SPs. Two possible examples: (*i*) *join*, using internal cluster migrations, increasing the *CPU* sharing; (*ii*) move task(s) to other clusters with free SPs.

The *ward* MP sends a message to the $SP_{candidate}$ to migrate the running task(s) to a new *SP*. As previously explained, the *ward MP* always chooses an *SP* from its cluster to receive the task(s) running on the $SP_{candidate}$. The operating system of the $SP_{candidate}$

executes the task migration. To execute the task migration, the OS sends a set of messages to the target *SP*:

- Task code;
- Task data: local data and *BSS* memory segments;
- Stack data: data stored in the memory corresponding to the stack;
- *TCB* (Task Control Block): a data structure that stores the task state, including the values stored at each register, Program Counter, Stack Pointer, size of the object code and data (for migration purposes);
- Message Requests: a structure with the received requests for messages;
- Pipe: all messages produced by the task but not yet delivered;
- Tasks location: addresses of the tasks that communicate with the task being migrated.

The first three messages refers to the memory contents of the task. It would be simpler to send the task's memory content to the target *SP*, but this would induce larger messages in the *NoC*, inducing a larger impact on other applications. For this reason, each memory segment is sent separately. The remaining messages are control messages, stored in the operating system, which are necessary for the correct execution of the task.

The target *SP* after receiving all messages related to the task migration, execute the following actions: (*i*) sends a message to all SPs that communicate with the migrated task with the new task address (6); (*ii*) sends a message to *ward* notifying that the migration process ended (7).

The task migration process induce an overhead in the application execution time when it occurs. The Results section evaluates this overhead.

5.6 Kernel Migration

The kernel (operating system) migration differs from task migration. While in task migration it is possible to optimize the amount of data to be transmitted, the kernel migration requires the transmission of complete memory contents from the faulty MP to the new position ($SP_{candidate}$).

Another difference between migration methods is the migration management. While in task migration the kernel itself performs this process, in the kernel migration this is not possible because the processor is faulty and isolated by wrappers. Thus, it was added in the

DMNI module the ability to treat specific packets, which start the process of transferring the memory content.

The first step of the kernel migration process is to prepare the $SP_{candidate}$ to receive the memory contents (code and data) of the faulty MP, MP_F. The healthy *ward* MP, MP_H, notifies the $SP_{candidate}$ that it will receive the kernel executing in MP_F, through a `wait_kernel_message`. A field in the packet header of this message defines that the *DMNI* module will process the message payload, not the processor. This message induces in the *PE* the following actions: (*i*) hold the processor and configure the *DMNI* module to write incoming packets into the memory, from address zero; (*ii*) after configuring the *DMNI* to write packets directly into the memory, the DMNI sends a `wait_kernel_acknowledge` message to MP_H.

Once received the `wait_kernel_acknowledge` message, the MP_H notifies MP_F to send the kernel to $SP_{candidate}$ through a `send_kernel_message`. The kernel migration is simpler than the task migration in the sense that only one message is transmitted with the complete memory contents, but induces congestion in the *NoC* during the kernel transmission.

Figure 5.9 presents how MP_F handles the `send_kernel_message`. The *DMNI* of the MP_F handles this message (1), transferring the memory contents to the $SP_{candidate}$ (2), using the data *NoC*. After transferring the memory contents, the *DMNI* is configured to avoid any transmission from the faulty processor, preventing Byzantine faults.

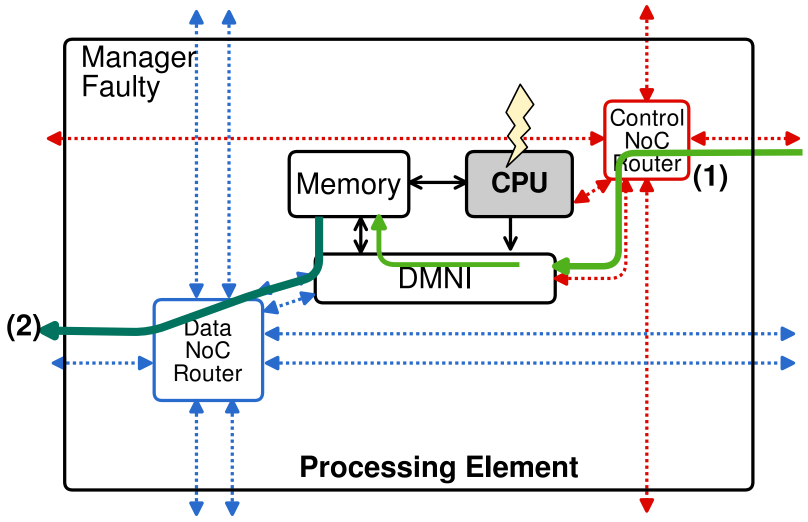


Figure 5.9 – Kernel migration process in a faulty *MP*.

5.7 Recovery Protocol

This section presents the complete protocol, using the mechanisms presented above, for *VGM* and *LM* faults.

5.7.1 Virtual Global Manager Fault Recovery

Figure 5.10 shows the execution of the recovery protocol, assuming a fault in the *VGM*. The example adopts a 6x6 many-core instance, with 3x3 clusters. The fault occurs during the application admission.

Figure 5.10(a) presents the system before the recovery protocol. The application injector starts the transmission of a new application to the cluster managed by *VGM* (these events are detailed in Chapter 6), events **1** to **7**. These events correspond to the transmission of the object code of tasks A, B, and C to their SPs. The fault in *VGM* is detected before the complete transmission of all tasks belonging to the new application - event **8**. As detailed in Section 5.2, the *VGM* broadcasts a `fail_CPU_message` to its *ward* (LM1). Two actions are taken:

1. All injectors interrupt the transmission of the current and future applications to the many-core. If an application is being injected in the many-core, this injection is interrupted. In this example, this corresponds to event **9**, by interrupting the transmission of task C.
2. The *ward MP*, LM1, initiates the process to recovery the *VGM*.

The first action executed by LM1 is to inject a `freeze` message to all SPs (event **10**). An *SP* receiving a `freeze` message verifies if exists a task managed by *VGM_F* (faulty *VGM*), and stops the tasks in a safe state. Any task managed by the *VGM_F* goes to a freeze state, avoiding its scheduling by the kernel (Section 5.4).

Next, LM1 sends a `send_kernel_message` to the *VGM_F* (event **11**). The *DMNI* handles this message, transferring the memory contents to *$SP_{candidate}$* using the data *NoC* (event **12**). After transferring the memory contents, the *DMNI* is configured to avoid any transmission from the faulty *VGM_F* processor. The *$SP_{candidate}$* after received the kernel (event **13**), restart the execution, now as a $VGM_{0,1}$ and send three messages:

1. `unfreeze` message to all SPs (event **14**);
2. `remove_task_message`, as an application admission was interrupted, the SPs needs to release the data structures related to the tasks belonging to application that was incompletely received (event **15**);

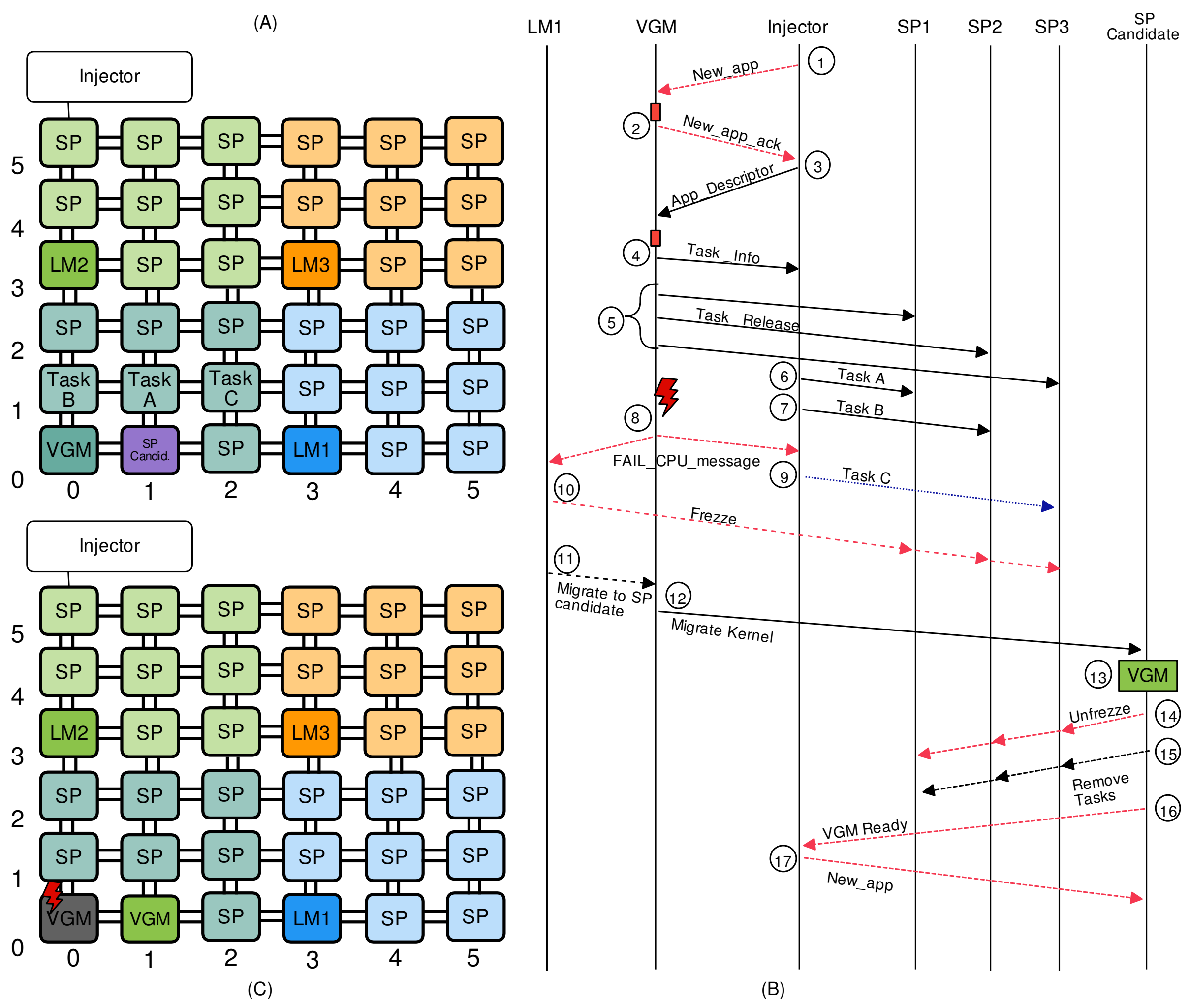


Figure 5.10 – A 6x6 instance of the reference many-core system with 3x3 clusters and a Virtual Global Manager migration.

3. The new *VGM* send a `vgm_ready` message to all injectors after the system recovery is complete (event 16).

The new *VGM* sends the `remove_task_message` to the tasks belonging to the application that was interrupted. Their address were defined during the mapping process. This message resets the TCBs, in such way to release the memory page.

After these steps, injectors are allowed to restart the application injection protocol (event 17). Figure 5.10 (c) presents the system after the recovery protocol, the new *VGM* is located at $PE_{0,1}$, and the SPs are free to receive new tasks.

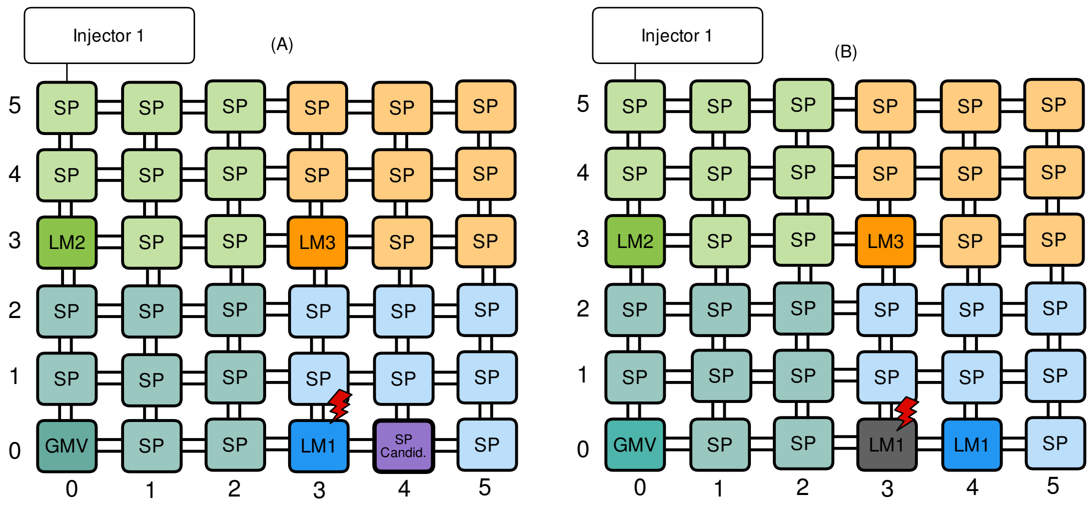


Figure 5.11 – Local manager migration.

5.7.2 Local Manager Fault Recovery

Figure 5.11 presents a scenario handled by the recovery protocol when a fault is detected in an *LM*. The protocol is similar to the previous one. In this scenario, the LM1 cluster has an application with four tasks and four free SPs. The main difference is that Injectors are allowed to inject new applications into the system. Only the cluster managed by the faulty *LM* freezes its execution, and cannot receive new applications. The injector restarts the application injection protocol when it receives the notification that the *LM* becomes faulty. The *VGM* does not select the cluster to receive new applications up to the completion of the faulty *LM* migration.

Figure 5.12 details the steps to recover the system from a fault in an *LM*. Cluster 1 receives application mapping requests (**1**), assigning tasks to four SPs in its cluster. In this example, four SPs execute at least one task and four *SP* are free as showed in Figure 5.11(a). After assigning the tasks' location in the cluster, $SP_{5,0}$ is elected as a new $SP_{candidate}$ (Section 5.3), and $LM1$ notifies its *ward*, *VGM*, that $SP_{5,0}$ is the $SP_{candidate}$, located at $SP_{5,0}$ and has no task assigned to it (**2**). At a given moment (**3**), a fault is detected in $LM1$. The *brNoC* receives the fault notification, and broadcast a `fail_CPU_message`. The first action, after the fault notification message, is to broadcast a `freeze` message (Section 5.4) to all tasks managed by $LM1$ (**4**).

The *ward MP*, *VGM*, starts the kernel migration. Events **5** to **8** correspond to the kernel migration protocol (Section 5.5): prepare the $SP_{candidate}$ to receive the kernel (**5**), acknowledgment message to the *ward* MP (**6**), notification to transmit the kernel (**7**), transmission of the kernel (**8**). Once the kernel received, the $SP_{candidate}$ restarts, assuming the role of a new LM1. After restarting, the new LM1 sends an *unfreeze* message to the stopped

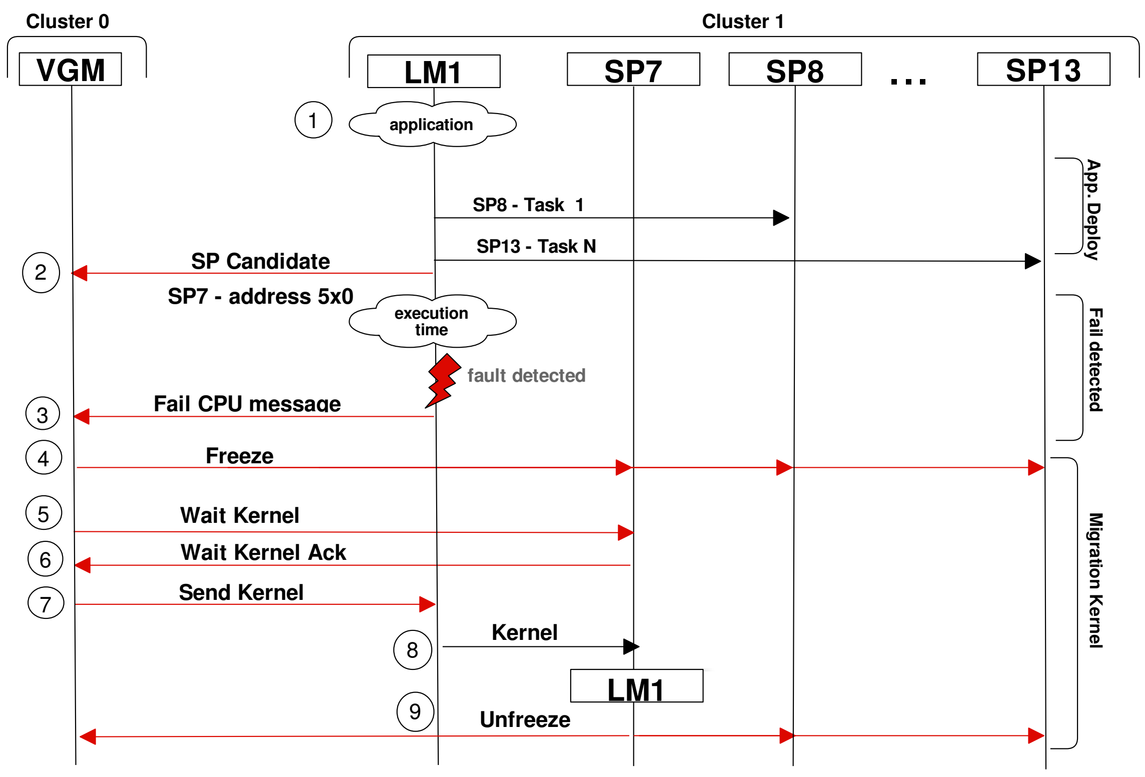


Figure 5.12 – Local manager recovery protocol.

tasks (9). This message unfreezes the tasks managed by the new *LM*1 and also transmits the new LM1 address to the SPs.

5.8 Final Remarks

This Chapter presented an original method for recovering the management functions of a many-core in the presence of permanent faults in a manager processor, *LM* or *VGM*. It is worth remembering that permanent faults in these management processors can isolate an entire region of the many-core (a cluster), or even prevent its use if the fault occurs in the *VGM*. Thus, the methods proposed in this Chapter increase the many-core lifetime at the cost of a reduction in overall system performance, given that the wrappers isolate processors with permanent faults. Thus, the presence of the single point of failure is eliminated internally to the many-core, since the functions associated with the *VGM* can migrate to another processor, allowing the many-core to continue to receive applications from the "injector" peripheral. However, the presence of only one application injector connected to the system still represents a single point of failure, since this peripheral is responsible for requesting and transmitting applications to the many-core. If the injector or its connection to the many-core fails, the many-core will be disconnected from the external environment. The next Chapter discusses two solutions for tolerating these faults. The first is related to the redundancy of the links between the injector and the many-core, and the second one is the redundancy of injectors.

6. APPLICATION ADMISSION RECOVERY METHOD

The previous Chapter presented the recovery method at the system level, by migrating a kernel to some *PE*. This Chapter presents the recovery method when a new application is entering into the system, and a fault occurs during this process.

Section 6.1 presents modifications carried-out at the *MCSoC*s boundaries to cope with faults during application admission, including multiple injectors and redundant links. Section 6.2 shows the admission protocol, without the presence of faults. Section 6.3 details the recovery method during application admission. Section 6.4 concludes this Chapter.

6.1 Injector hardware MCSoCs and faults mechanims

The injector communicates with the *VGM* through an external *NoC* border to inject a new application in to the system. Figure 6.1 details two major modifications added to the system: (*i*) redundant links between a injector and the *MCSoC*; (*ii*) multiple injector modules.

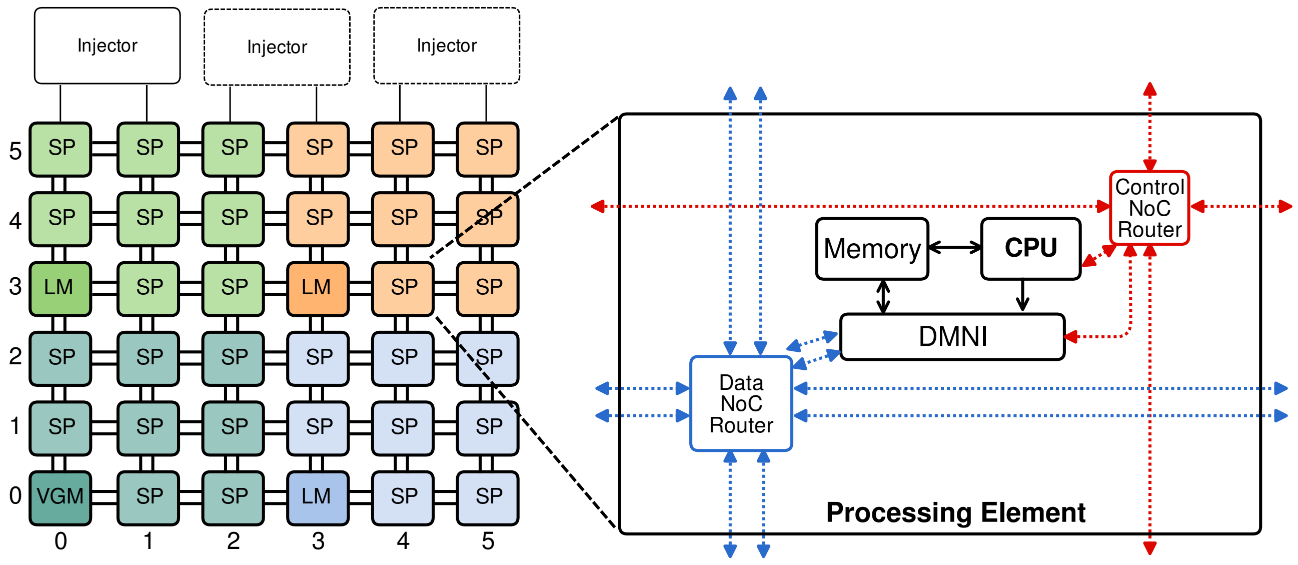


Figure 6.1 – MCSoCs with multiple injector instances and redundant links.

Each injector has now a primary and a secondary link connected to the *MCSoC*. The primary connection is the default communication port with the *MCSoC*, and the secondary link is only used when a fault is detected in the primary connection. Thus, the redundant connection between the injector and the *MCSoC* prevents that faults in links do not block the application admission from this injector.

Multiple injector instances may be connected to the MCSoCs boundary's (e.g., each injector can be seen as an Ethernet port). Injectors can insert new applications at any time, respecting the protocol with the *VGM*. The admission protocol can handle permanent faults, at runtime, in the links with the *MCSoC*. Also, the injector may deal with fault notification in the *VGM* or *LM* during application admission. Briefly, after a message from *VGM* or *LM* reporting a fault, the injector aborts the application admission and waits for the manager to restart. If a fault is detected in the injector link the injector change to the secondary link.

Each injector inserts new applications into the system independently. The injectors use the Control NoC to synchronize the application insertion protocol with the *VGM*. The Data NoC is used only to send the Task object code to the SPs.

Figure 6.2 presents the fault detection module ([Fochi et al., 2015]) in the routers and their connection to an injector. A fault module in the router detects the faulty link and notifies the injector (f0 and f1). The default link between the injector and the *MCSoC* is the primary connection. When a permanent fault is detected in the primary connection, the injector switches the communication to the secondary connection. To complete this change, the injector sends a message in broadcast to all PEs and managers notifying the new injector port address. An injector is disconnected from the *MCSoC* when a fault arises in the secondary link, but the system continues to operate due to the presence of other injectors.

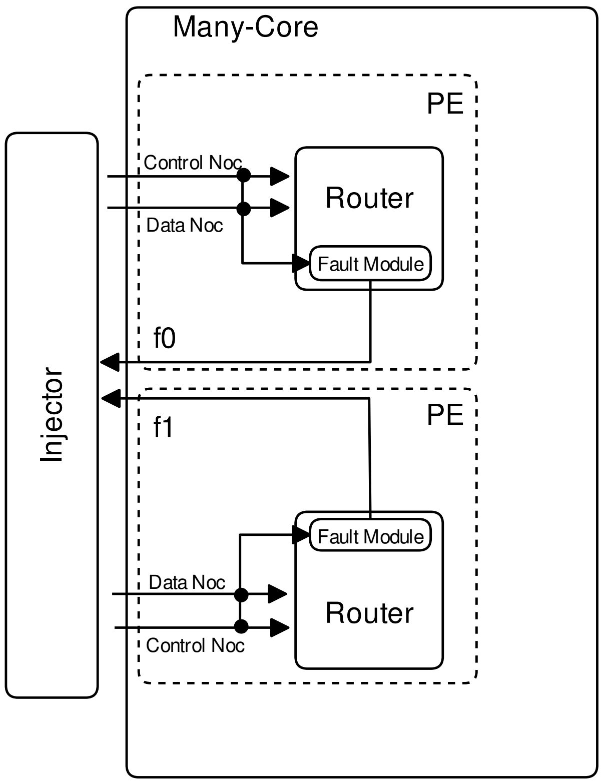


Figure 6.2 – Interface between a injector and the MCSoC.

6.2 Application Admission Protocol

System management demands procedures such as application admission, handling of monitoring messages, and control of resources through actions such as task migration and commands to change the voltage and frequency (*DVFS*) of some PEs. The management is distributed in clusters, in such a way that this process does not become a bottleneck in large systems. This section addresses application admission management, which enables the injection of new applications into the system. This presentation does not consider the presence of faults, subject detailed in the next Section.

Figure 6.3 presents the application admission protocol. At the system startup, the Virtual Global Manager (*VGM*) notifies all Injectors that it is ready to receive new applications (event **1** in Figure 6.3). All Injectors connected to the system receive the `VGM_READY_MESSAGE` message (**2**). When an Injector has an application to be executed in the *MCSoC*, it sends a `NEW_APP_MESSAGE`, having in its the payload the the application's tasks number (**3**). The *VGM* receives the application request, select the cluster according to some heuristic, and returns a `NEW_APP_ACK_MESSAGE` (**4**) to the Injector that made the request. This message contains the manager ID responsible to receive and map the application. The messages exchanged in these four initial steps use the *brNoC* (broadcast transmission mode) since the *VGM* may be located at any physical position of the system. The `NEW_APP_MESSAGE` is only consumed and treated by the *VGM*.

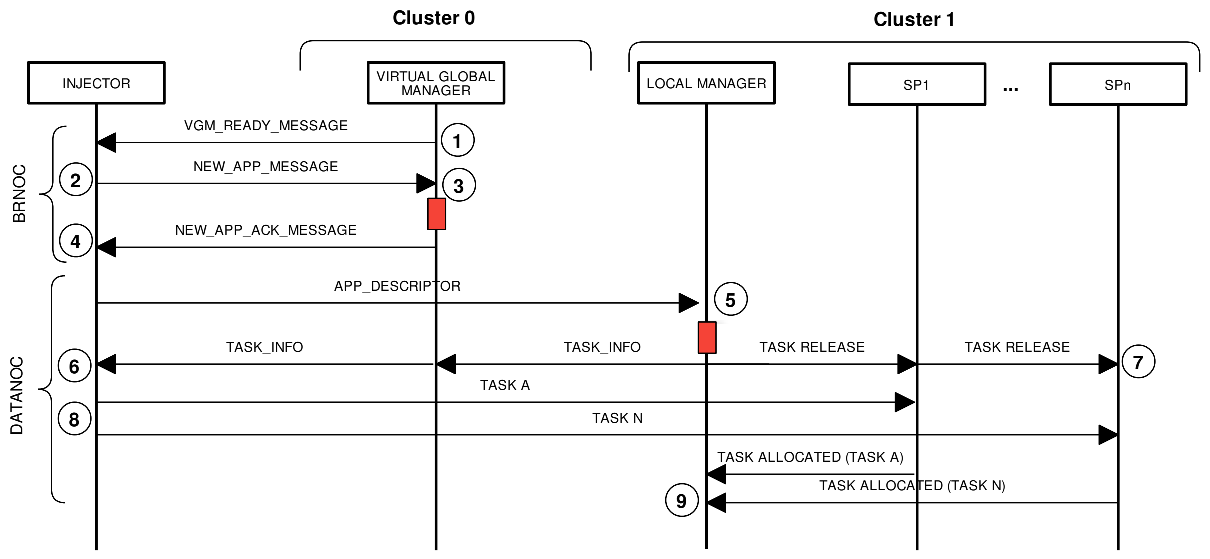


Figure 6.3 – Application Admission Protocol.

Next, the Injector sends the `APP_DESCRIPTOR` (**5**) message to the selected manager. This message contains the application description. Each application task is represented by a tuple $\{task_{ID}, text_{size}, data_{size}, bss_{size}\}$. The $task_{ID}$ is a unique identifier for the task, while the other parameters are used for task migration, corresponding the memory size of each object code segment. The APP_DESCRIPTOR message uses the data *NoC*, due to the message size. Once received this message, the manager PE executes a mapping heuristic [Castilhos et al., 2016],[Marcon et al., 2017], selecting the addresses to receive the tasks. After executing the mapping heuristic, the manager sends two types of messages:

- TASK_INFO: message sent to the Injector, with the tuple $\{task_{ID}, task_{address}\}$, address obtained during the mapping process (**6**).
- TASK_RELEASE: message sent to all SPs that will receive tasks (**7**). This message contains the task control information (size of the memory segments) for a specific task (**7**), and notifies the SP that it will receive a task to execute.

For each received TASK_INFO message, the Injector sends the object code of the task to the address embodied in the message (**8**). When an *SP* receives and stores the object code in the *PE* internal memory, the task is scheduled to execute, and it sends a TASK_ALLOCATED message to its manager *PE* (**9**). Once received all TASK_ALLOCATED messages, the admission protocol finishes. A fault in a *VGM* or an *LM* in any steps of this protocol is critical. If it occurs, the injector needs to restart the application admission protocol.

6.3 Fault Recovery during Application Admission

This section presents the actions executed during a permanent fault detected in the primary link of the Injector, and the actions executed when a fault arises in a manager *PE* during the application admission.

6.3.1 Recovery from a Fault in the Primary Link

Figure 6.4 presents the application admission protocol with a fault in the primary link. The injector starts the application admission protocol (events **1** to **4**). At a given moment (**5**), a fault is detected in the primary link, and the injector is transmitting Task B object code (**6**) when the fault occurs. The injector sends to the *SP*, through the *brNoC*, a Broken_Task_Message, notifying that the transmission was interrupted. The SP2 receives this message (event **7**), and release the memory area used to receive the task. Next, the SP2 kernel sends a Broken_Task_ACK message, notifying the injector that SP2 is ready to receive Task B object code (event **8**). The injector broadcasts a Change_Link_Service to the *VGM* and all managers, notifying the new connection address. After that, the injector retransmits Task B object Code. The protocol continues its operation, now using the secondary link.

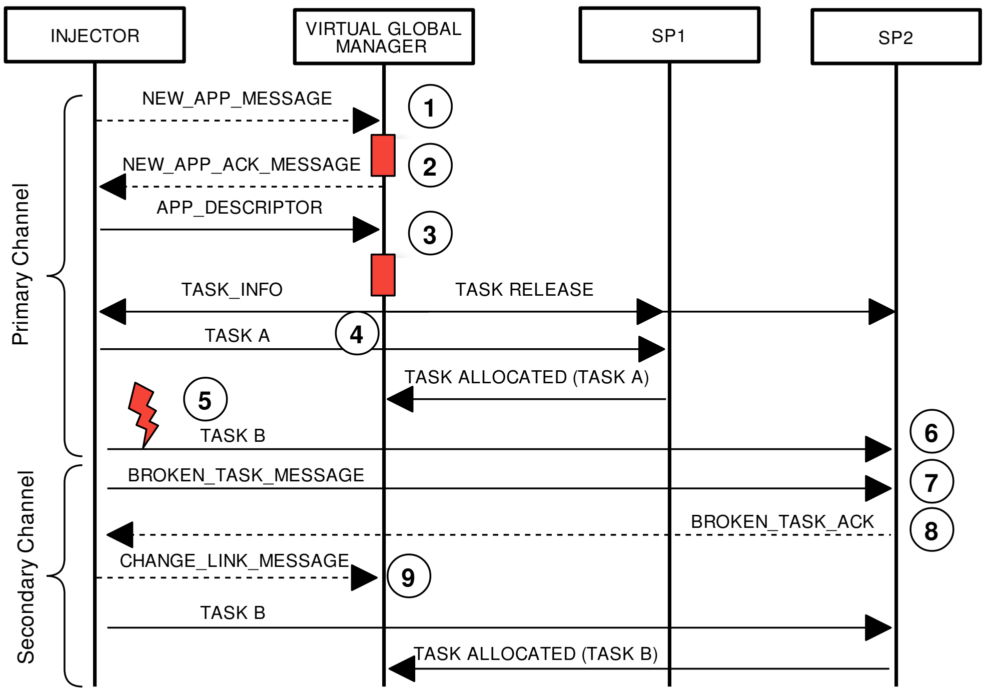


Figure 6.4 – Fault in a the primary link.

If the fault is detected when the injector is not sending object codes, i.e., the data *NoC* is not in use, the injector simply switches the link and sends a `Change_Link_Message` message (**9**) to the *VGM* and all managers, notifying the new connection address.

6.3.2 Recovery from a Fault in a Manager PE During Application Admission

Figure 6.5 presents the actions taken by an Injector when it is notified about a permanent fault in a Manager *PE*.

Fault in the *VGM*. When a permanent fault is detected in the *VGM*, a fault notification is broadcasted to its *ward*, being also received by all injectors. If an application admission is occurring in the *VGM* cluster, the application admission is interrupted, and all injectors will wait for the new *VGM* address. Otherwise it is verified if an application admission is occurring in another cluster. In such a case, the admission continues, since the *VGM* cluster is not required at this step of the protocol. Once a new *PE* receives the *VGM*, the injectors are notified, and the admission of new applications may restart.

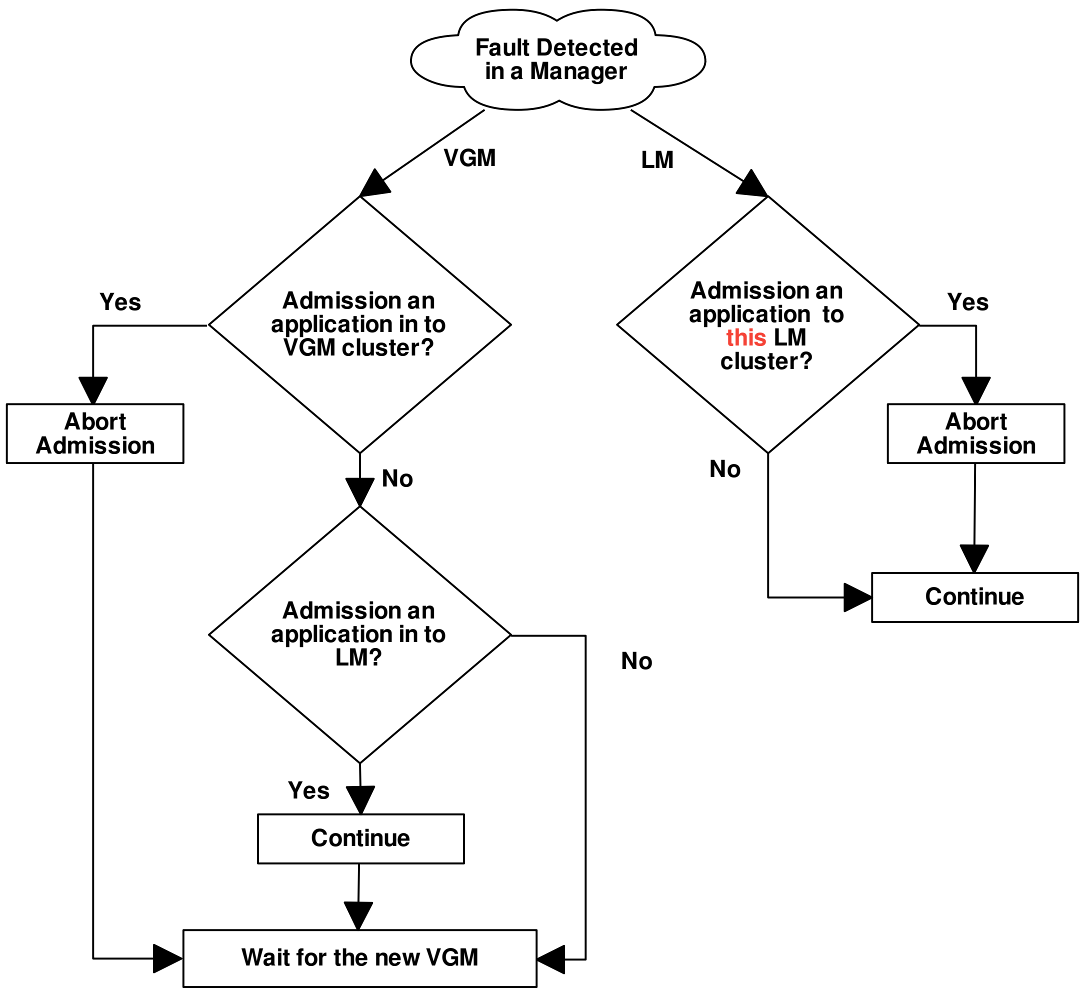

Fault Detected in a Manager
VGM
LM
Admission an application in to VGM cluster?
Yes
Abort Admission
No
Admission an application in to LM?
Yes
Continue
No
Wait for the new VGM
Admission an application to this LM cluster?
Yes
Abort Admission
No
Continue

Figure 6.5 – Recovery from a fault in a manager *PE* during application admission.

Fault in an *LM*. When a permanent fault is detected in a Local Manager the injector and the *VGM* identify the faulty *LM* by the broadcast fault notification. If an Injector is sending an application to the cluster where the *LM* is faulty, the application admission is aborted. In this case the injector re-initiates the application admission protocol, with the *VGM* selecting another cluster. The *VGM* continues to admit new applications, excluding the faulty cluster up to the migration of the faulty *LM* to a new address.

6.4 Final Remarks

This Chapter presented another important contribution of this Thesis, fault tolerance between the *MCSoC* and the external components responsible for injecting new applications. The foundation of the method is redundancy, in the injector links and in the number of injectors. The proposed method avoids single points of failure, allowing the *MCSoC* to continue to receive new applications in the presence of faults in the links of a given injector; as well as with disconnected injectors, given the redundancy of the same. Thus, with these techniques, it is concluded the part related to the development of fault tolerance techniques at the management level in *MCSoC* systems. The next Chapter evaluates the set of techniques proposed in this Chapter, as well as the techniques of the previous Chapter.

7. EXPERIMENTAL RESULTS

This Chapter presents results related to the methods proposed in Chapters 5 and 6. Experiments are executed using a clock-cycle accurate *RTL* SystemC model of the reference many-core platform. Applications and kernel are described in C language, compiled from C code and executed over the platform model. The experiments adopt a 6x6 many-core instance, organized in 3x3 clusters. To evaluate the recovery protocol, five benchmarks execute in the MCSoC: MPEG decoder (5 tasks), Prod Cons (2 tasks), DTW (6 tasks), Synthetic (6 tasks) and Dijkstra (6 tasks). Figure 7.1 presents the benchmarks' task graphs.

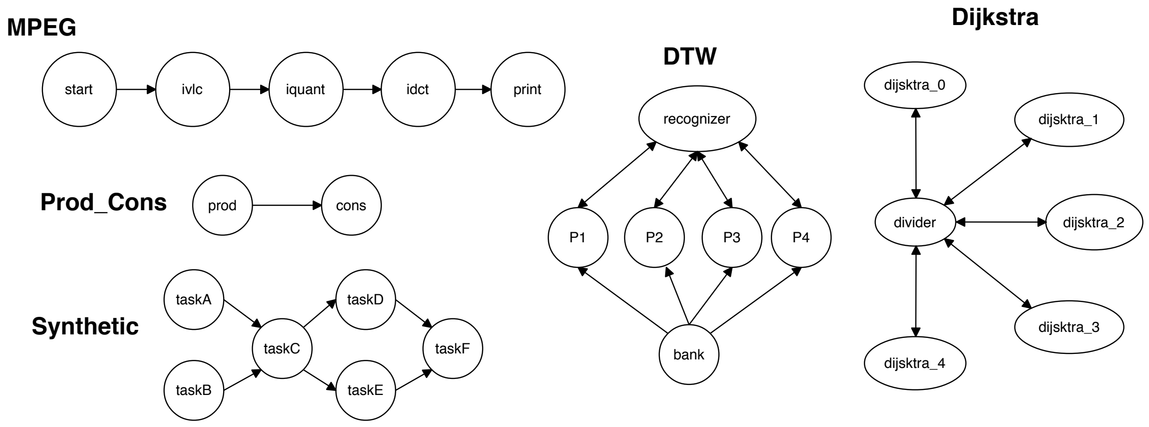


Figure 7.1 – Task graphs used in the experiments.

This Chapter evaluates the Workload Execution Time (*WET*) and the recovery method overheads, in milliseconds (@100MHz). A common overhead in the experiments is the time required to migrate the kernel (64 KB), 1.5 ms (average value), and the time to migrate one task (10 KB, code and data), 0.3 ms (average value). These overheads vary proportionally with the kernel and task sizes. The reason to keep the same kernel and task sizes in the experiments comes from the fact that they do not impact in the remaining protocol steps.

7.1 Recovery Results from a Fault in a Manager

This section presents a scenario when the fault is injected in the *VGM* or *LM*. This first evaluation corresponds to the best-case scenario for the protocol, since the *VGM* or the *LM* is not receiving an application and the $SP_{Candidate}$ is free (without any task assigned to it).

Figure 7.2 presents the test case to recover the *VGM*. Figure 7.2(a) presents the *MCSoC* state before the recovery method, being $SP_{2,2}$ the $SP_{Candidate}$. When the fault is detected by the LM1 (*VGM ward*), the manager recovery method starts. Figure 7.2(b) presents the system state after the *VGM* migration to the $SP_{Candidate}$(2,2).

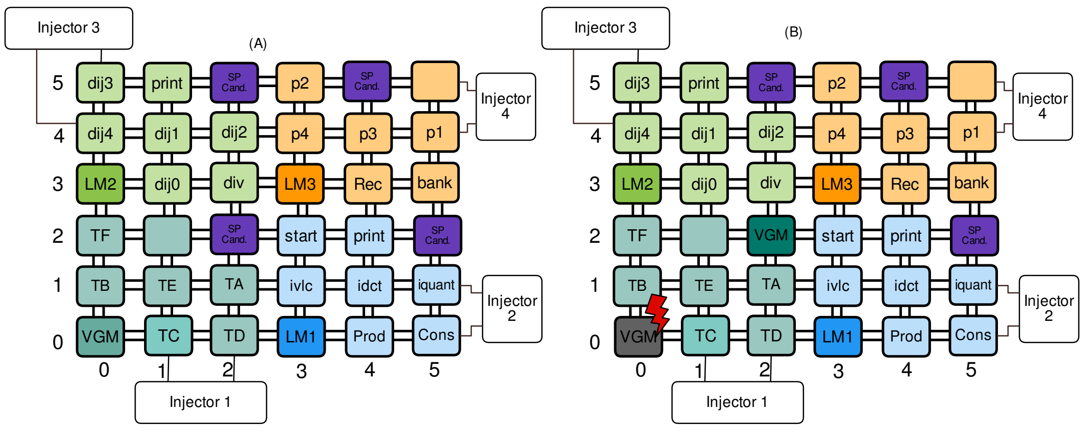


Figure 7.2 – Recovery method for the *VGM*.

Table 7.1 details the time spent at each recovery protocol step. The 1st line contains the time when a fault was inserted and detected, 2.8 ms. The 2nd line shows the moment when the kernel migration starts. The 4th line corresponds to the moment when the recovery ended, 4.32 ms. The difference, 1.52 ms, is the delay mentioned above to migrate the kernel.

The last two lines correspond to the *WET*, with (5th) and without (6th) the recovery method. The difference (1.78 ms) is slightly higher than the kernel recovery time. The reason for explaining the increase in the total execution time is mainly due to the rescheduling of tasks. Figure 7.3 illustrates the task scheduling of the Synthetic application, which was running in the left-most cluster. It is possible to observe the moment when all tasks were suspended due to the freeze message, and later the moment of reactivation (unfreeze message). Given the interdependence between the tasks, there is an overhead for the resynchronization between them.

Table 7.1 – Overhead - VGM recovery.

	Time (ms)
Fail CPU	2.8
Freeze	2.81
Wait Kernel	2.97
Unfreeze	4.32
WET (with recovery)	10.13
WET (baseline)	8.35

Table 7.2 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
DTW	0.00	8.32	1
Synthetic	0.00	10.08	1
Dijkstra	1.50	7.98	4
Mpeg	2.54	7.04	3
Prod_cons	2.17	5.30	3

Table 7.2 shows the time when each application starts and ends its execution. Note that all applications were executing when the fault was injected into the *VGM*. The t_{start} in Table 7.2 corresponds to the moment that an Injector must start the application injection into the *MCSoC*. The *VGM* execute the cluster selection, in the sequence occurs the task mapping, the transmission of the object code of the tasks to the SPs, and finally, the task is

scheduled. Thus, even if $t_{start} = 0$, as for the Synthetic application, this application actually starts at 0.5 ms.

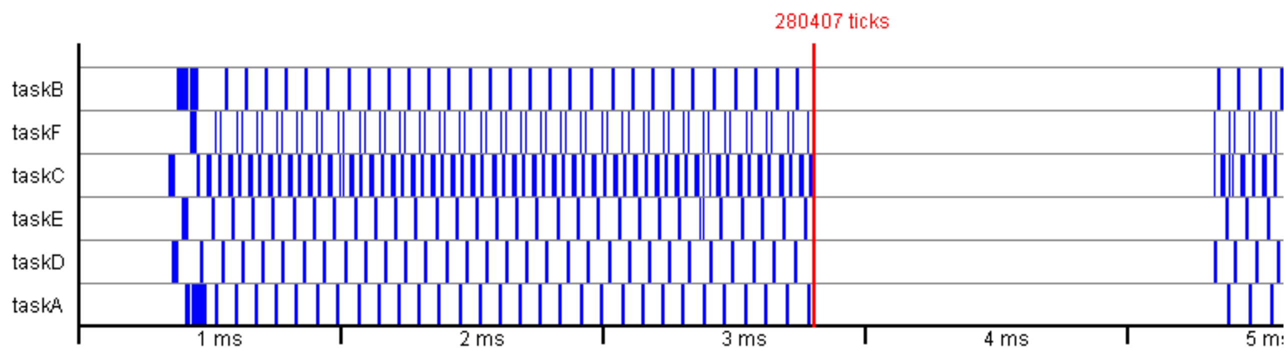


Figure 7.3 – Scheduling of the Synthetic tasks, showing the moment when the application is suspended.

Figure 7.4 presents the test case to recover an *LM*. Figure 7.4(a) presents the *MCSoC* state before the recovery method, being $SP_{5,2}$ the $SP_{Candidate}$. When the fault is detected by the *VGM* (LM1 *ward*), the manager recovery method starts. Figure 7.4(b), presents the system state after the LM1 migration to the $SP_{Candidate}$(5,2).

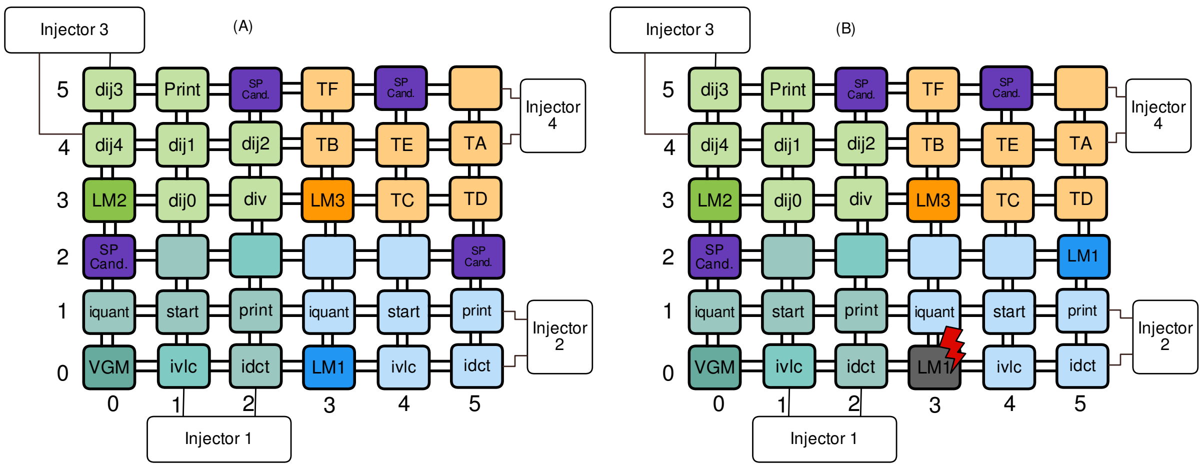


Figure 7.4 – Recovery method for the LM1.

The result is similar to the *VGM* recovery, 1.52 ms to migrate the LM1 kernel. Table 7.3 details the time spent at each recovery protocol step. Table 7.4 shows the time when each application starts and ends its execution. The *WET*, with and without recovery, is the same. The reason for explaining the same *WET* is that the application affected during the recovery method (MPEG-1) finishes its execution before the Synthetic application (9.05 ms). The overhead occurs only in the MPEG-1 execution time, 5.01 ms to 6.64 ms, resulting in an overhead equal to 1,63 ms.

Table 7.3 – Overhead – LM recovery.

	Time (ms)
Fail CPU	2.5
Freeze	2.51
Wait Kernel	2.67
Unfreeze	4.02
WET (with recovery)	9.06
WET (baseline)	9.06

Table 7.4 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
MPEG-1	2.00	8.54	4
Synthetic	0.00	9.05	1
Dijkstra	1.50	7.09	4
MPEG-2	0.00	5.06	1

In both scenarios, with faults injected into the *VGM* or LM1, the overhead is the time to migrate the memory contents (code and data of the faulty manager) to the $SP_{Candidate}$. When an MP fails, the tasks it manages should be suspended to prevent control messages from being lost (freeze), delaying applications. For both *VGM* or *LM* recovery, the overhead was the same, corresponding to 1.5 ms@100MHz, or 150,000 clock cycles.

7.2 Recovery Results from a Fault in a Manager With Task Migration

This section presents a scenario when the fault is injected in the *VGM* or *LM*, and the cluster has all resources in use. Thus, the $SP_{Candidate}$ is not free. i.e., it has tasks assigned to it, being necessary to execute task migration before the recovery method starts.

Figure 7.5 shows the test case to recover the *VGM*, executing task migration before kernel migration. Figure 7.5(a) presents the *MCSoC* state before the recovery protocol, being $SP_{1,0}$ the $SP_{Candidate}$. When the fault is detected by LM1 (*VGM ward*), the manager recovery method starts. The task *TC* migrates from $SP_{1,0}$ to $SP_{5,2}$. After task migration, the $SP_{Candidate}(1,0)$ receives the *VGM* kernel. Figure 7.5(b) presents the system state after the *VGM* kernel migration.

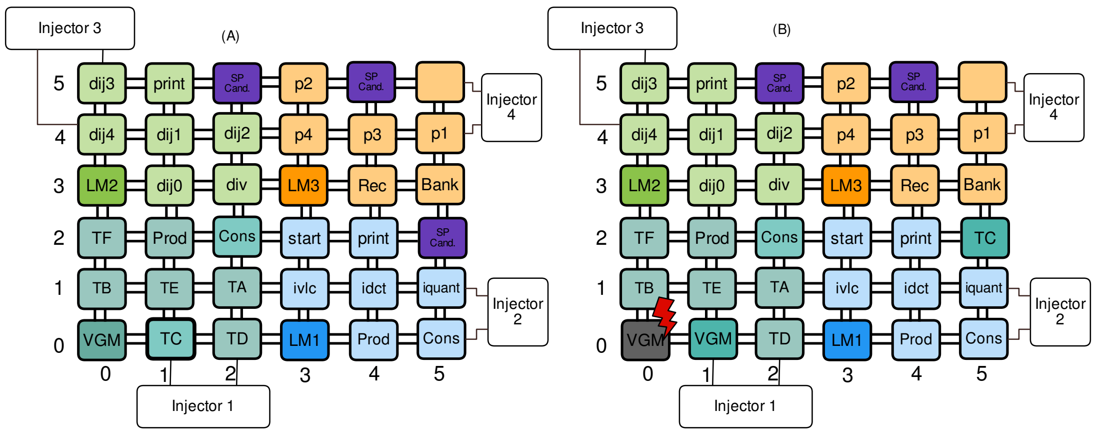

(A)
(B)
Injector 3
Injector 4
Injector 2
Injector 1
dij3 print SP Cand. p2 SP Cand.
dij4 dij1 dij2 p4 p3 p1
LM2 dij0 div LM3 Rec Bank
TF Prod Cons start print SP Cand.
TB TE TA ivlc idct iquant
VGM TC TD LM1 Prod Cons
TF Prod Cons start print TC
VGM VGM TD LM1 Prod Cons
0 1 2 3 4 5

Figure 7.5 – Recovery method for the VGM and a task migration.

Figure 7.6 illustrates the task scheduling of *TC*, which was running in the $SP_{1,0}$ and migrated to $SP_{5,2}$. It is possible to observe the moment when the task was suspended due to the freeze message, migrated and later the moment of reactivation (unfreeze message).

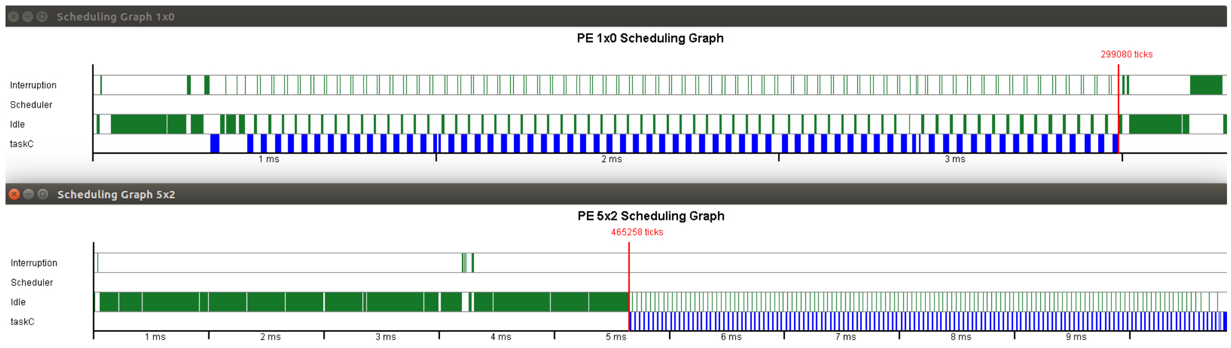


Figure 7.6 – Scheduling of Task C, showing the moment when the task migrate.

Table 7.5 details the time spent at each recovery protocol step. The 1st and 2nd lines present when the fault was inserted and detected, 3.0 and 3.01 ms, respectively. The 3rd line shows the moment when task migration ended. The 4th line shows the moment when kernel migration starts. The 5th line shows the moment when the recovery ended. Table 7.6 shows the time when each application starts and ends its execution.

The overhead induced by the recovery method and a task migration was 1.65 ms. However the *WET* with recovery presents a overhead equal to 1.31 ms. The overhead is lower than expected due to the fact that task *TC* is originally in a position with high data traffic, and with migration, its mapping reduced the network congestion. This experiment shows that a reduced number of hops between tasks, the primary function of the mapping heuristics, may impact negatively in the application performance.

Table 7.5 – Overhead - VGM recovery and task migration.

	Time (ms)
Fail CPU	3.00
Freeze	3.01
Migration	3.29
Wait Kernel	3.30
Unfreeze	4.65
WET (with recovery)	9.92
WET (baseline)	8.61

Table 7.6 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
DTW	0.00	8.33	1
Synthetic	0.30	9.89	1
Dijkstra	1.50	7.11	4
Mpeg	2.50	7.04	3
Prod_cons	2.17	5.24	4
Prod_cons	2.00	7.01	3

Figure 7.7 presents the test case to recover LM1 with task migration, being the result similar to the previous one. Figure 7.7(a) presents the *MCSoC* state before the recovery method, being $SP_{4,0}$ the $SP_{Candidate}$. When the fault is detected by the *VGM* (LM1 *ward*), the manager recovery method starts. The $SP_{2,2}$ receives task *TC* from the $SP_{4,0}$. After task

migration, the $SP_{Candidate}$(4,0) receives the LM1 kernel. Figure 7.7(b), presents the system state after LM1 kernel migration.

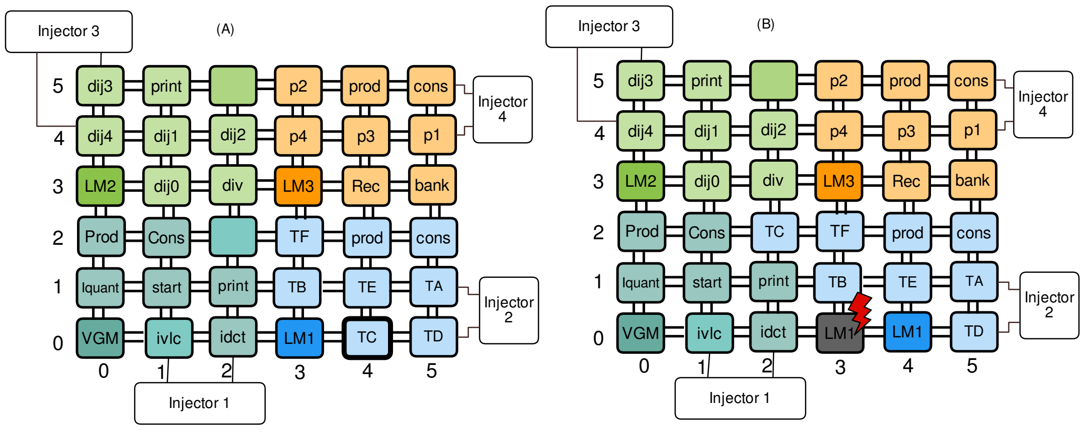


Figure 7.7 – Recovery method for the LM1 and a task migration.

Table 7.7 details the time spent at each recovery protocol step. The 1st and nth2 lines present when the fault was inserted and detected, 3.0 and 3.01 ms, respectively. The 3rd line shows the moment when task migration ended. The 4th line shows the moment when kernel migration starts. The 5th line shows the moment when the recovery ended. Table 7.8 shows the time when each application starts and ends its execution. In this experiment, the overhead induced by the recovery method and a task migration was 1.65 ms, and the *WET* overhead 1.66 ms. They are, in practice, the same because the affected application by the LM1 fault is the one with the longest execution time.

Table 7.7 – Overhead - LM recovery and task migration.

	Time (ms)
Fail CPU	3.00
Freeze	3.01
Migration	3.29
Wait Kernel	3.30
Unfreeze	4.65
WET (with recovery)	11.86
WET (baseline)	10.20

Table 7.8 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
Synthetic	2.50	11.82	2
Mpeg	0.03	5.07	1
DTW	0.00	8.21	1
Dijkstra	1.50	7.10	4
Prod_cons	2.17	5.29	3
Prod_cons	2.00	5.12	4
Prod_cons	2.00	7.02	2

In both *VGM* or LM1 fault scenarios, the overhead is the time spend to migrate the memory contents (code and data of the faulty manager) to the $SP_{Candidate}$ and the task migration. When an MP fails, the tasks it manages should be suspended to prevent control messages from being lost (freeze). The freezing process delays the application. For both *VGM* or LM1 recovery and the task migration, the time overhead was 1.65 ms or 165,000 clock cycles.

7.3 Recovery Results from a Fault in the Primary Link

This section evaluates the recovery method when a fault is injected in the primary link of an injector. The first evaluation does not require task transmission, and the second one does. Figure 7.8 shows the test case to recover from a fault in the primary link without retransmission. This evaluation corresponds to a best-case scenario, since the Injector is not sending a task through the link.

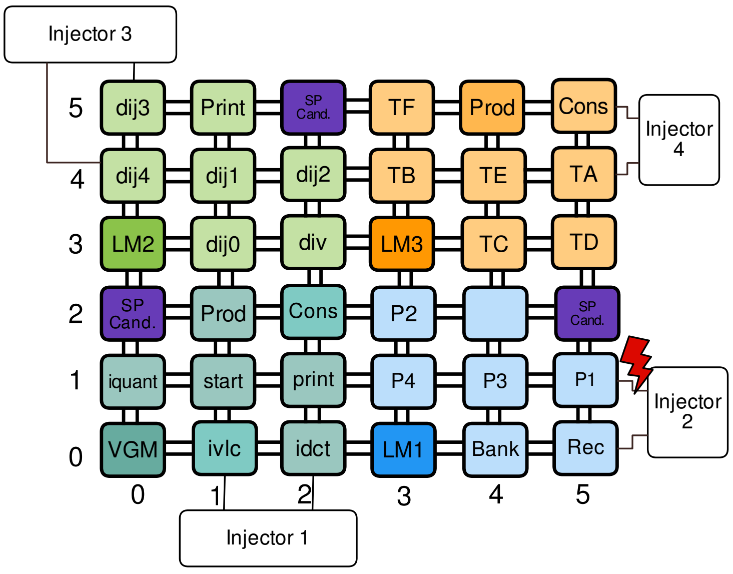


Figure 7.8 – Fault in a the primary link from Injector 2.

Table 7.9 details the time spent at each recovery step. The 1st line presents when the fault was inserted and detected – 0.8 ms. The second line presents when the Injector produces the `Change_Link_Message`, transmitted in broadcast by the *brNoC*. The overhead to change the link is 0.0011 ms (110 clock cycles), corresponding to the time required by the *brNoC* to broadcast a message. Table 7.10 presents the applications' execution time and the injector for each application. The *WET* is not affected because the fault occurred when the Injector was not sending a task.

Table 7.9 – Recovery overhead from a fault in the primary link.

	Time (ms)
Fail Link	0.8000
Change link	0.8011
WET (with recovery)	8.93
WET (baseline)	8.93

Table 7.10 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
Synthetic	0.5	8.61	1
Dtw	1.0	8.90	2
Mpeg	0.0	5.08	1
Dijkstra	1.5	7.08	3
Prod_Cons	2.0	5.15	4
Prod_Cons	2.0	5.28	3

Figure 7.9 presents a recovery scenario from a fault in the primary link with task retransmission. The scenario injects a fault in the primary link of Injector1 when it is transmitting task *TC* to $SP_{4,3}$.

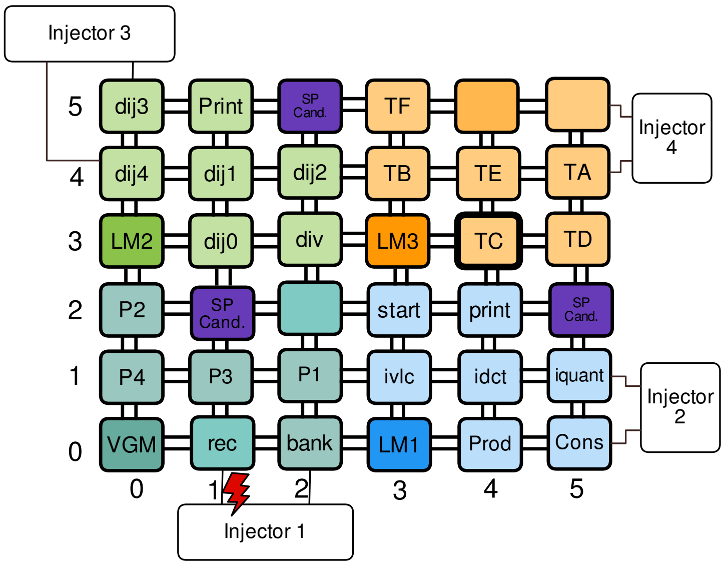


Figure 7.9 – Fault in a the primary link from Injector 1.

Table 7.11 details the time spent at each recovery step. The 1st line presents when the fault was inserted in the Injector1 primary link – 0.7367 ms. The 2nd line presents when the `Broken_Task_Message` is injected to $SP_{4,3}$, notifying it the incomplete task transmission. The 3rd line presents when the *brNoC* broadcasts the `Change_Link_Message`. The 4th line presents the time when Injector1 receives the `Broken_Task_ACK_Message` from $SP_{4,3}$. The 5th line present when the task *TC* retransmission starts, and the 6th line when task *TC* started at $SP_{4,3}$. Table 7.12 presents the applications' execution time and the injector for each application.

Table 7.11 – Recovery overhead from a fault in the primary link with task retransmission.

	Time (ms)
Fail Link	0.7367
Broken Task	0.7379
Change link	0.7395
Broken Task ACK	0.7450
Resend Task	0.7516
Task C Started	0.7690
WET (with recovery)	8.72
WET (baseline)	9.08

Table 7.12 – Applications' execution time.

Application	t_{start} (ms)	t_{end} (ms)	Injector
Synthetic	0.50	8.62	1
Mpeg	2.00	7.17	3
DTW	0.00	7.78	1
Dijkstra	1.50	7.07	4
Prod_cons	2.50	4.95	3

The overhead in this scenario is between the fault injection and the time to start the task that was interrupted due to the faulty link. The overhead induced by the recovery link process was in this experiment 0.0323 ms. Surprisingly, the WET of the scenario with fault recover ended before the baseline scenario. A hypothesis to explain this behavior is because the Synthetic application has an intensive communication profile. Delaying task C, data are produced and stored by tasks A and B. When task C starts, there is already data for it to consume and send to tasks D and E, without waiting for the generation of new data by tasks A and B.

7.4 Recovery Results from a Fault in a Manager During Application Admission

This section presents a scenario when the fault is injected in the *VGM*/*LM*, during an application admission. The cluster has a free resource ($SP_{Candidate}$) to receive the Manager. Thus it is not necessary to execute task migration before the recovery method starts.

Figure 7.10 shows the test case to recover the *VGM*. Figure 7.10(a) presents the *MCSoC* state when the fault is injected in the *VGM*, interrupting the admission of the Synthetic application. The Injector stops the application admission and waits for the *VGM* recovery. Figure 7.10(b) presents the *MCSoC* after the recovery method. When the *VGM* restarts, it releases the SPs that received tasks belonging to the interrupted application (Synthetic). Note that this application is now mapped in the cluster managed by the LM3 because the *VGM* admitted first the DTW application.

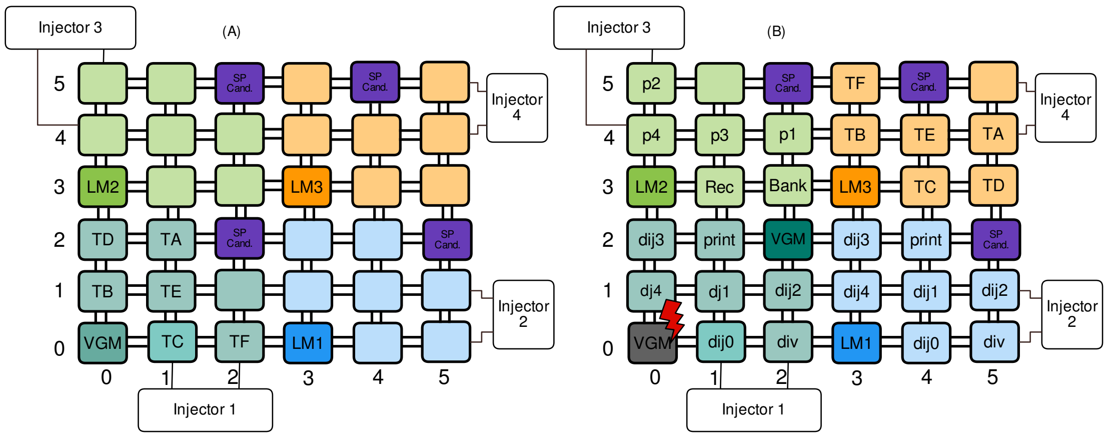


Figure 7.10 – Recovery method in a Manager during a application admission.

Table 7.13 details the time spent at each recovery step. The 1st and 2nd lines present when the fault was inserted and detected, 0.39 and 0.40 ms, respectively. The 3rd line shows the moment when kernel migration starts. The 4th line shows the moment when the recovery ended. Table 7.14 shows the time when each application starts and ends its execution.

In this experiment, the overhead induced by the recovery method was 1.57 ms and the *WET* overhead 1.29 ms. As the fault was injected at the beginning of the simulation, the Synthetic, DTW, and one Dijkstra applications were affected. The Table 7.14 presents when the applications should start and the actual time they started. These applications started after 2 ms because the *VGM* was recovered at 1.96 ms.

Table 7.13 – Overhead to a VGM recovery.

	Time (ms)
Fail CPU	0.39
Freeze	0.40
Wait Kernel ACK	0.58
Unfreeze	1.96
WET (with recovery)	10.22
WET (baseline)	8.93

Table 7.14 – Application's execution time.

Application	t_{start}/t_{real} (ms)	t_{end} (ms)	Injector
Synthetic	0.00/2.24	10.12	1
DTW	1.50/2.21	9.86	2
Dijkstra-1	1.00/2.31	8.49	1
Dijkstra-2	2.00/2.66	8.13	3

Figure 7.11 presents the test case to recover a *LM* (LM3). Figure 7.11(a) presents the *MCSoC* before the recovery method, being $SP_{5,5}$ the $SP_{Candidate}$. When the fault is detected by the LM2 (LM3 *ward*), the manager recovery method starts. Figure 7.11(b), presents the system state after the LM3 migration to the $SP_{Candidate}$(5,5).

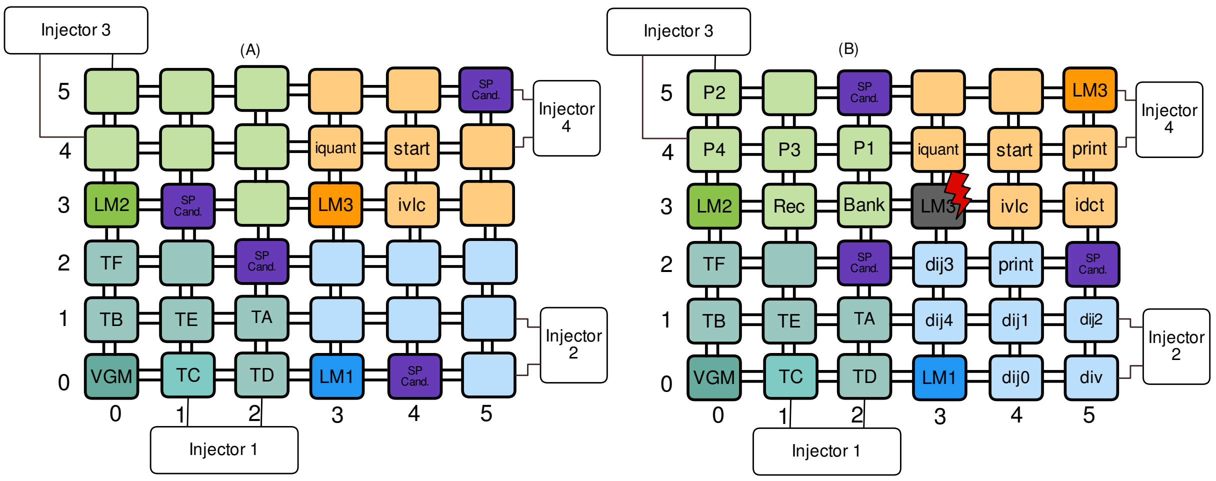


Figure 7.11 – Recovery method in a Manager during a application admission.

Table 7.15 details the time spent at each recovery protocol step. Table 7.16 shows the time when each application starts and ends its execution. The result is similar to the *VGM* recovery, 1.62 ms to migrate the LM3 kernel. The *WET*, with and without recovery is the same. The reason for explaining the same *WET* is that the application affected during the recovery method (MPEG) finishes its execution before the Synthetic application (8.55 ms). The overhead only affects the MPEG execution time.

Table 7.15 – Overhead from a LM recovery.

	Time (ms)
Fail CPU	1.29
Freeze	1.30
Wait Kernel ACK	1.53
Unfreeze	2.91
WET (with recovery)	8.93
WET (baseline)	8.93

Table 7.16 – Application's execution time.

Application	t_{start}/t_{real}	t_{end} (ms)	Injector
Synthetic	0.00/0.02	8.55	1
DTW	1.00/1.24	8.90	2
MPEG	1.00/3.14	7.94	1
Dijkstra	2.00/2.31	8.07	3

7.5 Final Remarks

Table 7.17 summarizes the results presented in this Chapter. It is possible to state that a failure in a manager processor (*VGM* or *LM*) induces a runtime overhead of around 1.5 ms (150,000 clock cycles), and it increases according to the size of the kernel memory footprint. The overhead due to the failure of the external links is minimal (100 to 3,200 clock cycles), ensuring that the *MCSoC* continues to operate even in the presence of faults in the communication with the external world.

Table 7.17 – Summary of Results.

Fault Location	**Relevant protocol feature**	**Protocol overhead** (kernel: 64KB/task: 10 KB)
VGM	**without** task migration	1.5 ms
LM		
VGM	**with** task migration	1.65 ms
LM		
Injector primary link	without task retransmission	0.001 ms
	with task retransmission	0.032 ms
VGM	Fault during application admission	1.57 ms
LM		1.62 ms

The evaluation made in this Chapter focused on the method overhead in terms of performance. There are two implementation costs: software and hardware. The cost of the software refers to the increase in memory required by the kernels running on *VGM*/*LM*, from 12 to 43 KB, and on Kernel Slave, from 19 to 34 KB.

The hardware costs associated with the methods can be listed as follows: (*i*) *brNoC* network, area equivalent to 20% of a data network router; (*ii*) wrappers, it require only logic gates to isolate control signals; (*iii*) it is assumed that the memory is protected by *ECC* (error-correcting codes) and that the network interface has access to this memory in case of processor failure. Therefore, the hardware cost is minimal, being portable for other MCSoCs architectures.

8. CONCLUSION

The Introduction of the Thesis declared as Thesis Statement the following paragraph:

> *The Thesis herein proposed aims to demonstrate that it is possible to develop a distributed MCSoC architecture, supporting permanent faults at critical points of the system, as in the processors executing management functions, and at the interface of the MCSoC with external entities responsible for deploying new applications into the system.*

When starting the process of defining the fault recovery methods, it was observed that hardware support was needed to allow fast communication with a broad set of processors simultaneously. The result was the *brNoC*, which allowed reaching a broad set of processors using broadcast communication. The *brNoC* is a generic *NoC* that can be used for multiple purposes. This work employed the BrNoC for fault-tolerance and system management purposes. As shown in Section 8.2, *brNoC* was also the basis for works related to security in many-cores.

The second observation related to the proposal for fault-tolerant methods in distributed architectures refers to the method of transferring applications to the *MCSoC*. Thus, a new method was proposed, based on *application injectors*. This method approximated the reference architecture to actual many-cores and was adopted in the baseline architecture.

These previous developments, *brNoC* and injectors, paved the way for the demonstration of the first part of the hypothesis: support for permanent faults at critical points of the system, as in a manager processor (*MP*).

This Thesis considered two options to keep the management state of the system after a fault in a *MP*: to maintain the management state in redundant data structures, or to use a monitoring scheme between *MP*s, in which a healthy *MP* isolates the faulty *MP*, starting the recovery process. The first option was discarded due to the excess of redundant messages and high memory consumption. The second method, monitoring between *MP*s, was then chosen.

Initially, it was adopted a method based on the response time to determine fault in a *MP* (*ward messages*). This method, although efficient, would generate a very long response delay for fault detection, which could compromise the content of the *MP* memory. Given the development of the *brNoC*, it was chosen to use it for the fault notification given the reduced time for the transmission of the fault notification.

The proposed method is an original contribution of this work because it does not require that the management data remain in redundant structures. The *MP* that presents a permanent fault is isolated by the healthy *MP*, being the content of its memory migrated to another processor. This new processor assumes the role of *MP*, with the management context stored in the original *MP*. There is a global loss of performance because the many-core will have one less processor, but it will continue to operate despite the permanent fault in the processor.

An important premise for the developed proposal is that memory content can be migrated after fault detection and processor isolation. It is suggested as future work the monitoring of transient faults in the *MP*. After a certain amount of faults, this *MP* migrates to another processor. This processor would be in a "quarantine state", running user applications. When reaching a threshold in the number of faults, the processor would be finally isolated from the system.

The results obtained demonstrated the correctness of the first part of the hypothesis, through the fault injection in local and global *MP*s, with the system always operating correctly again after the *MP* migration.

The second part of the hypothesis is related to the interface of the *MCSoC* with external entities responsible for deploying new applications into the system. In this case, the work adopted two levels of redundancy. Local redundancy, at the link level, and injector redundancy. The first redundancy level allows a given injector to continue to send new applications to the system if the primary connection with the system fails. The second redundancy level ensures that even if a given injector becomes isolated, there is still the possibility of sending applications to the system with a second injector.

Thus, we conclude that the original hypothesis is demonstrated, with the proposal of an architecture having distributed and fault-tolerant management, with no single point of failure. The methods proposed increase the many-core lifetime at the cost of a reduction in overall system performance, given that the wrappers isolate processors with permanent faults. Thus, the presence of the single point of failure is eliminated internally to the many-core, since the functions associated with the *VGM* can migrate to another processor, allowing the many-core to continue to receive applications from the injector peripherals.

The software cost of the proposed methods corresponds to about 258% in the manager kernel and 78% in the kernel slave. The hardware costs associated with the methods can be enumerated as follows: (*i*) *brNoC* network, area equivalent to 20% of a data network router; (*ii*) wrappers, it require only logic gates to isolate control signals; (*iii*) it is assumed that the memory is protected by *ECC* (error-correcting codes) and that the network interface has access to this memory in case of processor failure. Therefore, the hardware cost is minimal, being portable for other MCSoCs architectures with similar architectural features

8.1 Future Works

As a guideline for future works, this Thesis has room for improvements as follow:

- Extend the method to cover faults in slave processors (*SP*s). Fault tolerance for *SP*s has a rich literature. A method can be developed to recovery the *SP* from a permanent fault in its main components: *NoC* router, processor, and memory. The main goal is to enable the fault recovery without re-execution. The recovery method developed in this Thesis can be applied to the *SP* with a fault detection module.
- Extend the method to cover transients faults, preventing faults in the manager processors. This work targeted only permanent faults. It is suggested as future work the monitoring of transient faults in the *MP*. After a certain amount of faults, this *MP* migrates to another processor. This processor would be in a "quarantine state", running user applications. When reaching a threshold in the number of faults, the processor would be finally isolated from the system.
- Evaluate the application feasibility of the proposed recovery method without using the control NoC – *brNoC*. The reasons to adopt the *brNoC* were advanced in the text, which is mainly the rapid notification of a large set of PEs by using broadcast communication. We consider that its possible an implementation of the recovery methdos without using a control *NoC*. An adaptation of the methods and the corresponding evaluation, only with the data *NoC* can be developed to evaluate the proposed methods with a standard NoC-based many-core system.
- Include a hardware module to detected the fault in the *CPU*. In this work, we assume that a fault detection module responsible for triggering the recovery method. Future work can implement a module to detect transient and permanent faults in the *CPU*.
- Extend the method to detect permanent faults in the boundaries of the manager *PE* (*NoC* links), migrating the manager *PE* to prevent its isolation to the remaining of the system.
- *brNoC* (control NoC) for multiple purposes. This work employed the *brNoC* for monitoring, notification, and system management services. Another work from the group used *brNoC* for security, path discovery and secure zones. Cache coherence and *QoS* can be explored with *brNoC*.

8.2 Publications

Table 8.1 presents the publications made during the Thesis, relating them to the respective Chapters, when applicable.

Table 8.1 – Summary of Publications.

Publication Reference	Relationship with the Thesis
Ruaro, M.; Caimi, L.; Fochi, V.; Moraes, F. **A Framework for Heterogeneous Many-core SoCs Generation** In: LASCAS, 2019	Chapter 3 - development of the Injectors
Fochi, V.; Caimi, L.; Silva, M.; Wachter, E.; Moraes, F. **Fault-tolerance at the Management Level in Many-core Systems** In: SBCCI, 2018	Chapter 5 - System Management with task migration
Fochi, V.; Caimi, L.; Ruaro, M.; Wachter, E.; Moraes, F. **System Management Recovery Protocol for MPSoCs**. In: SOCC, 2017, pp. 367-374.	Chapter 5 - Basis of the System Management with task migration.
Wachter, E.; Caimi, L.; Fochi, V.; Munhoz, D.; Moraes, F. **BrNoC: a Broadcast NoC for Control Messages in Many-core Systems.** Microelectronics Journal, Volume 68, October 2017, Pages 69–77.	Chapter 3 - development of the BrNoC
Wächter, E. W.; Fochi, V.; Barreto, F.; Amory, A.; Moraes, F. **A Hierarchical and Distributed Fault Tolerant Proposal for NoC-based MPSoCs** IEEE Trans. on Emerging Topics in Computing, 2016 (accepted), v.6(4), pp 524-537, Oct.-Dec. 2018.	Fault-tolerance at the MPSoC level, routers and processors. Transition between the MsC to the PhD.
Wächter, E. W.; Fochi, V.; Barreto, F.; Amory, A.; Moraes, F. **A layered approach for fault tolerant NoC-based MPSoCs** In: LATS, 2016	
Caimi, L.; Fochi, V.; Wachter, E.; Moraes, F. **Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications** In: LASCAS, 2018	Publications not directly related to the Thesis subject - use of the BrNoC and wrappers infrastructure in the security domain
Caimi, L.; Fochi, V.; Moraes, F. **Secure Admission of Applications in Many-Cores** In: ICECS, 2018	
Caimi, L.; Fochi, V.; Wachter, E.; Munhoz, D.; Moraes, F. **Activation of Secure Zones in Many-Core Systems with Dynamic Rerouting** In: ISCAS, 2017, pp. 144-147	
Caimi, L.; Fochi, V.; Wachter, E.; Moraes, F. **Secure Admission and Execution of Applications in Many-core Systems** In: SBCCI, 2017.	

REFERENCES

[Barreto et al., 2015] Barreto, F., Amory, A. M., and Moraes, F. G. (2015). Fault Recovery Protocol for Distributed Memory MPSoCs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 421–424.

[Benini and Micheli, 2002] Benini, L. and Micheli, G. (2002). Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78.

[Bhowmik et al., 2016] Bhowmik, B., Deka, J. K., Biswas, S., and Bhattacharya, B. (2016). On-line Detection and Diagnosis of Stuck-at Faults in Channels of NoC-based systems. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 4567–4572.

[Bolchini et al., 2013] Bolchini, C., Carminati, M., and Miele, A. (2013). Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems. *Journal of Electronic Testing: Theory and Applications*, 29(2):159–175.

[Boraten and Kodi, 2016] Boraten, T. and Kodi, A. K. (2016). Packet security with path sensitization for NoCs. In *Proceedings of the Design, Automation Test in Europe Conference (DATE)*, pages 1136–1139.

[Braak et al., 2010] Braak, T. D. T., Burgess, S. T., Hurskainen, H., Kerkhoff, H. G., Vermeulen, B., and Zhang, X. (2010). On-line dependability enhancement of multiprocessor SoCs by resource management. In *Proceedings of the International Symposium on System (SSOC)*, pages 103–110.

[Caimi et al., 2018] Caimi, L., Fochi, V., Wachter, E., and Moraes, F. G. (2018). Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications. In *Proceedings of the IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4.

[Caimi et al., 2017a] Caimi, L., Fochi, V., Wachter, E., Munhoz, D., and Moraes, F. G. (2017a). Activation of Secure Zones in Many-core Systems with Dynamic Rerouting. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 144–147.

[Caimi et al., 2017b] Caimi, L., Fochi, V., Wachter, E., Munhoz, D., and Moraes, F. G. (2017b). Secure Admission and Execution of Applications in Many-core Systems. In *Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 65–71.

[Carara et al., 2009] Carara, E., de Oliveira, R., Calazans, N., and Moraes, F. G. (2009). HeMPS - a framework for NoC-based MPSoC generation. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1345–1348.

[Castilhos et al., 2013] Castilhos, G., Mandelli, M., Madalozzo, G., and Moraes, F. G. (2013). Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 153–158.

[Castilhos et al., 2016] Castilhos, G., Mandelli, M., Ost, L., and Moraes, F. G. (2016). Hierarchical Energy Monitoring for Task Mapping in Many-core Systems. *Journal of System Architecture*, 63(C):80–92.

[Chen et al., 2017] Chen, Y., Chang, E., Hsin, H., Chen, K., and Wu, A. (2017). Path-Diversity-Aware Fault-Tolerant Routing Algorithm for Network-on-Chip Systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):838–849.

[Cui et al., 2016] Cui, T., Li, J., Shafaei, A., Nazarian, S., and Pedram, M. (2016). An efficient timing analysis model for 6T FinFET SRAM using current-based method. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*, pages 263–268.

[Domingues et al., 2018] Domingues, A. R. P., Hamerski, J. C., and Amory, A. (2018). Broker Fault Recovery for a Multiprocessor System-an-Chip Middleware. In *Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

[Dutt et al., 2015] Dutt, N., Jantsch, A., and Sarma, S. (2015). Self-Aware Cyber-Physical Systems-on-Chip. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 46–50.

[Faruque et al., 2008] Faruque, M. A. A., Krist, R., and Henkel, J. (2008). ADAM: run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 760–765.

[Fick et al., 2009a] Fick, D., DeOrio, A., Chen, G., Bertacco, V., Sylvester, D., and Blaauw, D. (2009a). A Highly Resilient Routing Algorithm for Fault-tolerant NoCs. In *Proceedings of the Design, Automation Test in Europe Conference (DATE)*, pages 21–26.

[Fick et al., 2009b] Fick, D., DeOrio, A., Hu, J., Bertacco, V., Blaauw, D., and Sylvester, D. (2009b). Vicis: A reliable network for unreliable silicon. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 812–817.

[Fochi et al., 2018] Fochi, V., Caimi, L., da Silva, M. H., and Moraes, F. G. (2018). Fault-Tolerance at the Management Level in Many-Core Systems. In *Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

[Fochi et al., 2017] Fochi, V., Caimi, L., Ruaro, M., Wachter, E., and Moraes, F. G. (2017). System management recovery protocol for MPSoCs. In *Proceedings of the IEEE International System-on-Chip Conference (SOCC)*, pages 367–374.

[Fochi et al., 2015] Fochi, V., Wachter, E., Erichsen, A., Amory, A. M., and Moraes, F. G. (2015). An integrated method for implementing online fault detection in NoC-based MPSoCs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1562–1565.

[GAPH, 2018] GAPH (2018). Hardware Design Support Group. www.inf.pucrs.br/gaph/.

[Grecu et al., 2004] Grecu, C., Pande, P. P., Ivanov, A., and Saleh, R. (2004). Structured Interconnect Architecture: A Solution for the Non-scalability of Bus-based SoCs. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 192–195.

[Haghbayan et al., 2014] Haghbayan, M., Rahmani, A., Weldezion, A. Y., Liljeberg, P., Plosila, J., Jantsch, A., and Tenhunen, H. (2014). Dark Silicon Aware Power mManagement for Manycore Aystems under Dynamic Workloads. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 509–512.

[Heron et al., 2010] Heron, O., Guilhemsang, J., Ventroux, N., and Giulieri, A. (2010). Analysis of on-line self-testing policies for real-time embedded multiprocessors in DSM technologies. In *Proceedings of the IEEE International On-Line Testing Symposium (IOLTS)*, pages 49–55.

[Kamran et al., 2016] Kamran, A. et al. (2016). Stochastic Testing of Processing Cores in a Many-core Architecture. *Integration, the VLSI Journal*, 55(1):183–193.

[Kim et al., 2013] Kim, H., Vitkovskiy, A., Gratz, P. V., and Soteriou, V. (2013). Use it or lose it: Wear-out and Lifetime in Future Chip Multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 136–147.

[Knebel et al., 2016] Knebel, F., Rehman, S., Shafique, M., and Henkel, J. (2016). ageopt-rmt: Compiler-driven variation-aware aging optimization for redundant multithreading. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Li and Draper, 2016] Li, J. and Draper, J. (2016). Joint Soft-Error-Rate (SER) Estimation for Combinational Logic and Sequential Elements. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 737–742.

[Linder and Harden, 1991] Linder, D. H. and Harden, J. C. (1991). An Adaptive and Fault Tolerant Wormhole Routing Strategy for k-ary n-cubes. *Transactions on Computer*, 40(1):2–12.

[Marcon et al., 2017] Marcon, C., Webber, T., and Susin, A. A. (2017). Models of computation for NoC mapping: Timing and energy saving awareness. *Microelectronics Journal*, 60(1):129–143.

[Martins et al., 2016] Martins, A. L. M., Sant'Ana, A. C., and Moraes, F. G. (2016). Runtime energy management for many-core systems. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 380–383.

[Meloni et al., 2012] Meloni, P. et al. (2012). System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 517–524.

[Moraes et al., 2004] Moraes, F. G., Calazans, N., Mello, A., Moller, L., and Ost, L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93.

[Paul et al., 2015] Paul, J. et al. (2015). Self-adaptive Corner Detection on MPSoC Through Resource-aware Programming. *Journal of System Architecture*, 61(10):520–530.

[Reddy et al., 2016] Reddy, B., Vasantha, M., and Kumar, Y. (2016). A Gracefully Degrading and Energy-Efficient Fault Tolerant NoC Using Spare Core. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 146–151.

[Ruaro et al., 2019] Ruaro, M., Caimi, L., Fochi, V., and Moraes, F. G. (2019). A Framework for Heterogeneous Many-core SoCs Generation. In *Proceedings of the IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 89–92.

[Ruaro et al., 2016] Ruaro, M., Lazzarotto, F. B., Marcon, C. A., and Moraes, F. G. (2016). DMNI: A specialized network interface for NoC-based MPSoCs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1202–1205.

[Ruaro and Moraes, 2017] Ruaro, M. and Moraes, F. G. (2017). Demystifying the cost of task migration in distributed memory many-core systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4.

[Schedel et al., 2011] Schedel, K. et al. (2011). OctoPOS: a parallel operating system for invasive computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures*, pages 9–14.

[Silveira et al., 2016] Silveira, J., Marcon, C., Cortez, P., Barroso, G., ao M. Ferreira, J., and Mota, R. (2016). Scenario preprocessing approach for the reconfiguration of fault-tolerant NoC-based MPSoCs. *Microprocessors and Microsystems*, 40(1):137–153.

[Srinivasan et al., 2004] Srinivasan, J., Adve, S. V., Bose, P., and Rivers, J. A. (2004). The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages (177–186).

[Suraj Paul, 2018] Suraj Paul, Navonil Chatterjee, P. G. (2018). A permanent fault tolerant dynamic task allocation approach for Network-on-Chip based multicore systems. *Journal of Systems Architecture*, 97(1):287–303.

[Tajik et al., 2016] Tajik, H., Donyanavard, B., Dutt, N., Jahn, J., and Henkel, J. (2016). SPM-Pool: Runtime SPM Management for Memory-Intensive Applications in Embedded Many-Cores. *ACM Transactions on Embedded Computing Systems*, 16(1):25:1–25:27.

[Tsai et al., 2013] Tsai, W.-C., Chu, K.-C., Hu, Y.-H., and Chen, S.-J. (2013). Non-minimal, turn-model based NoC routing. *Microprocessors and Microsystems*, 37(8B):899 – 914.

[Tsoutsouras et al., 2017] Tsoutsouras, V., Masouros, D., Xydis, S., and Soudris, D. (2017). SoftRM: Self-Organized Fault-Tolerant Resource Management for Failure Detection and Recovery in NoC Based Many-Cores. *ACM Transactions on Embedded Computing Systems*, 16(5s):144:1–144:19.

[Veiga and Zeferino, 2010] Veiga, F. and Zeferino, C. A. (2010). Implementation of Techniques for Fault Tolerance in a Network-on-Chip. In *Proceedings of the Symposium on Computing Systems (SCS)*, pages 80–87.

[Vitkovskiy et al., 2012] Vitkovskiy, A., Soteriou, V., and Nicopoulos, C. (2012). A Dynamically Adjusting Gracefully Degrading Link-Level Fault-Tolerant Mechanism for NoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(8):1235–1248.

[Wachter et al., 2017] Wachter, E., Caimi, L. L., Fochi, V., Munhoz, D., and Moraes, F. G. (2017). BrNoC: A broadcast NoC for control messages in many-core systems. *Microelectronics Journal*, 68(1):69–77.

[Walters et al., 2011] Walters, J. P., Kost, R., Singh, K., , and Crago, S. P. (2011). Software-based fault tolerance for the Maestro many-core processor. In *Proceedings of the Aerospace Conference (AERO)*, pages 1–12.

[Wentzlaff et al., 2007] Wentzlaff, D. et al. (2007). On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31.

[Woszezenki, 2007] Woszezenki, C. (2007). Alocação de tarefas e comunicação entre tarefas em mpsocs. Master's thesis, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre.

[Yu et al., 2011] Yu, Q., Zhang, M., and Ampadu, P. (2011). Exploiting inherent information redundancy to manage transient errors in NoC routing arbitration. In *Proceedings of the Fifth ACM/IEEE International Symposium (IS)*, pages 105–112.

[Zhang et al., 2012] Zhang, Y., Morris, R., DiTomaso, D., and Kodi, A. (2012). Energy-Efficient and Fault-Tolerant Unified Buffer and Bufferless Crossbar Architecture for NoCs. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops PhD Forum (PHD)*, pages 972–981.

Appendices

A. INJECTOR FINITE STATE MACHINES

The Application Injector FSM is responsible for managing all Injector FSMs: Data NoC in, Data NoC out, BrNoC in, BrNoC out.

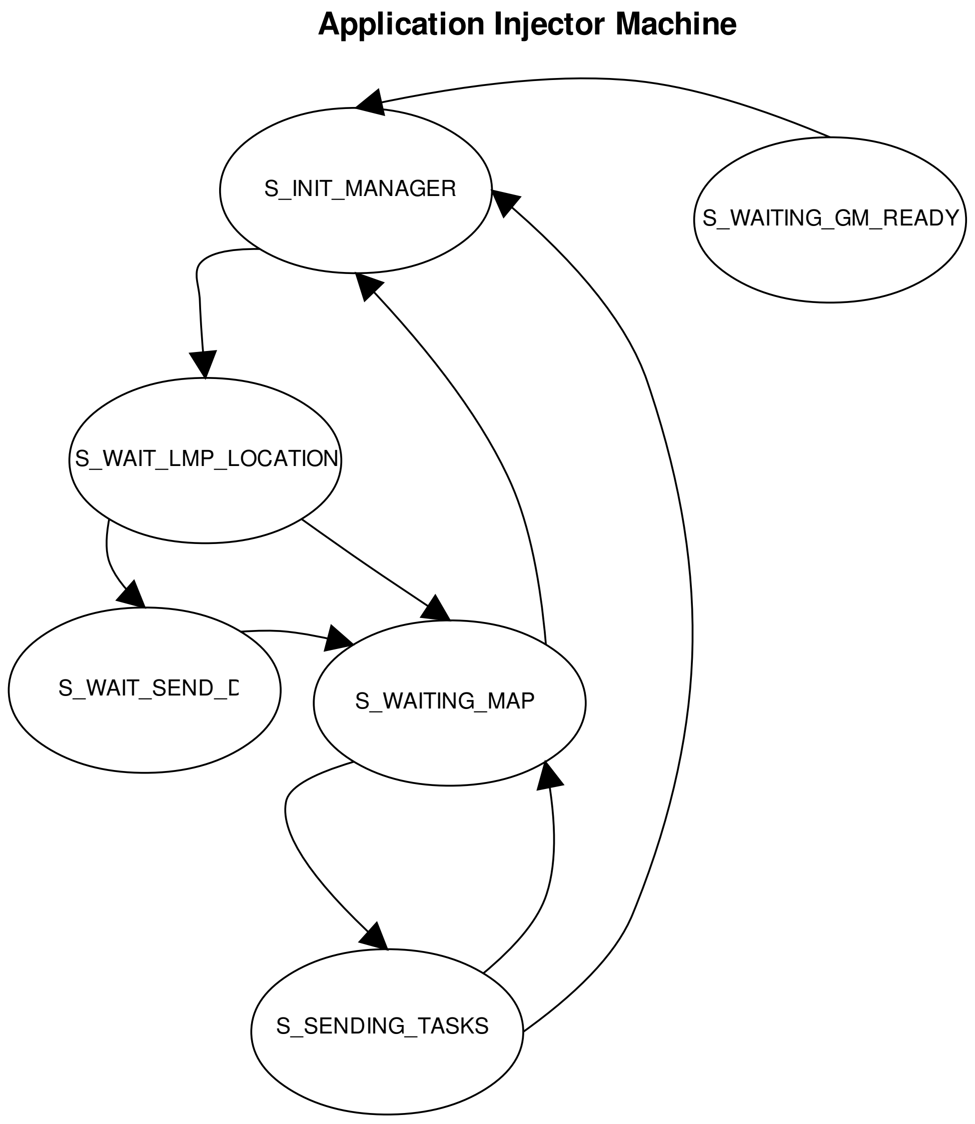


Figure A.1 – Application Injector FSM.

Data NoC Out FSM. This FSM sends data to the Data NoC of the MCSoC.

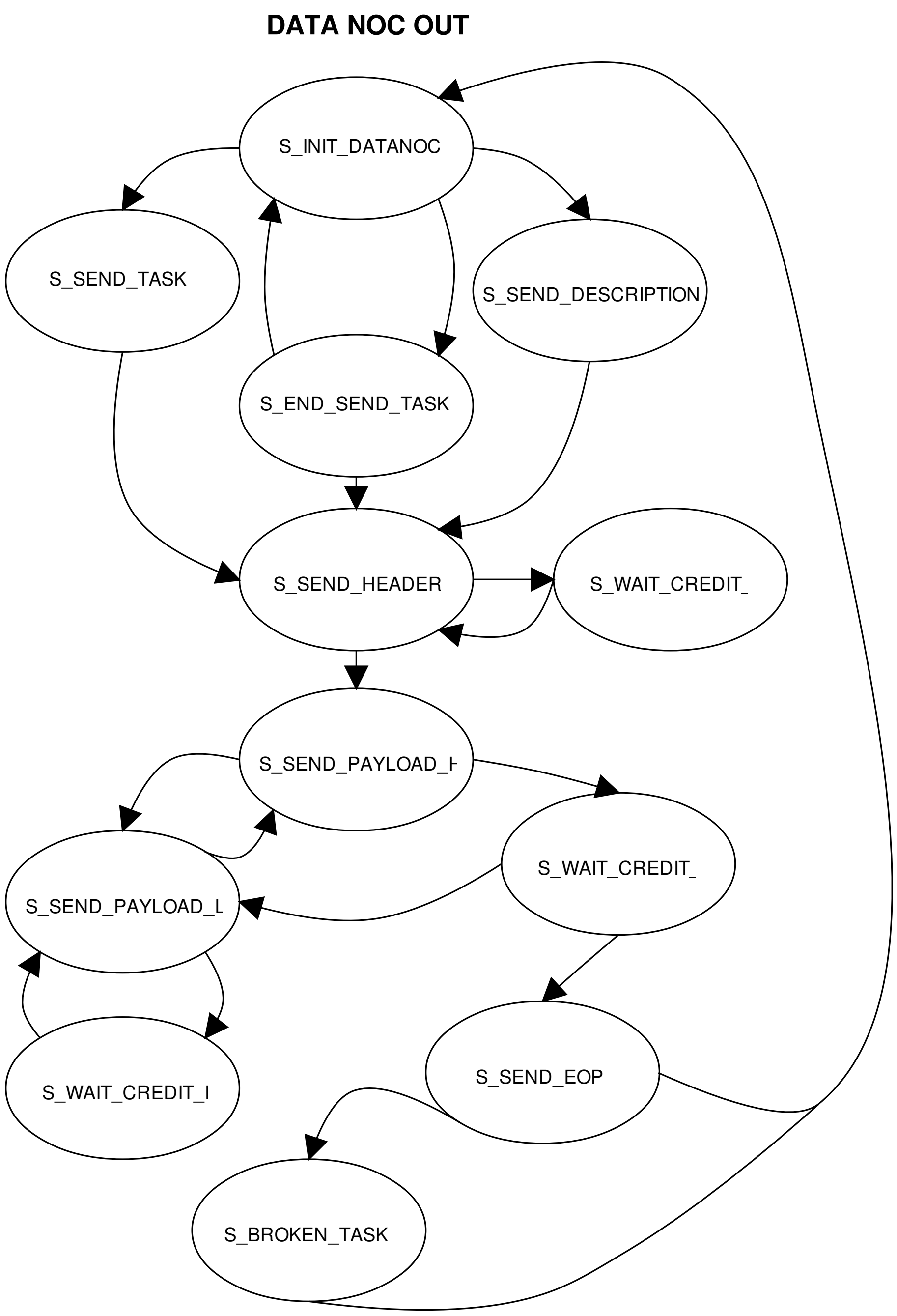


Figure A.2 – Data NoC Out FSM.

Data NoC In FSM. This FSM receives data from the Data NoC of the MCSoC.

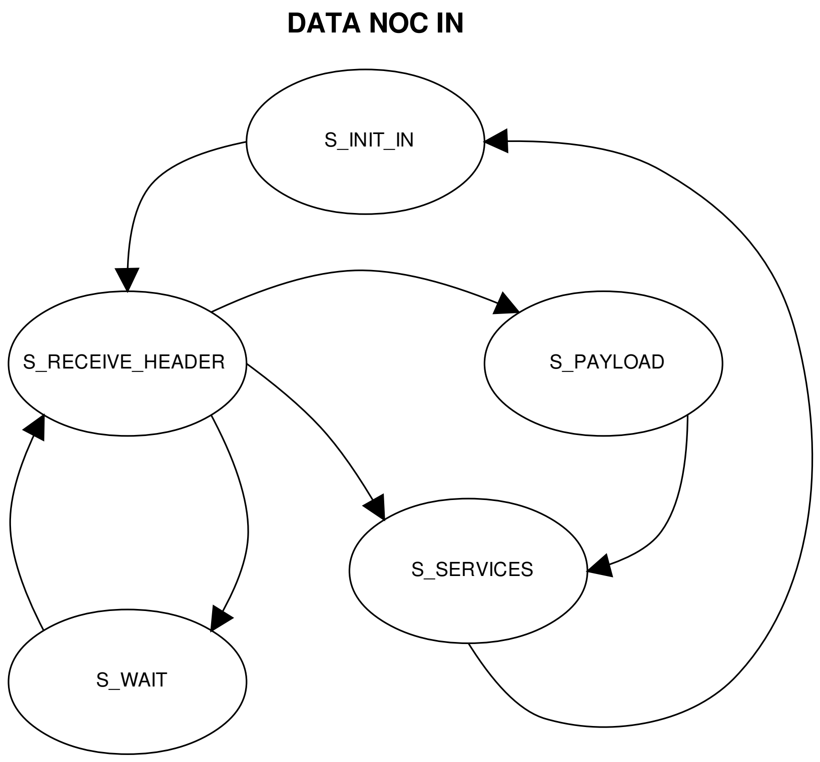


Figure A.3 – Data In NoC FSM.

BrNoC out FSM. This FSM sends packets to the control NoC of the MCSoC.

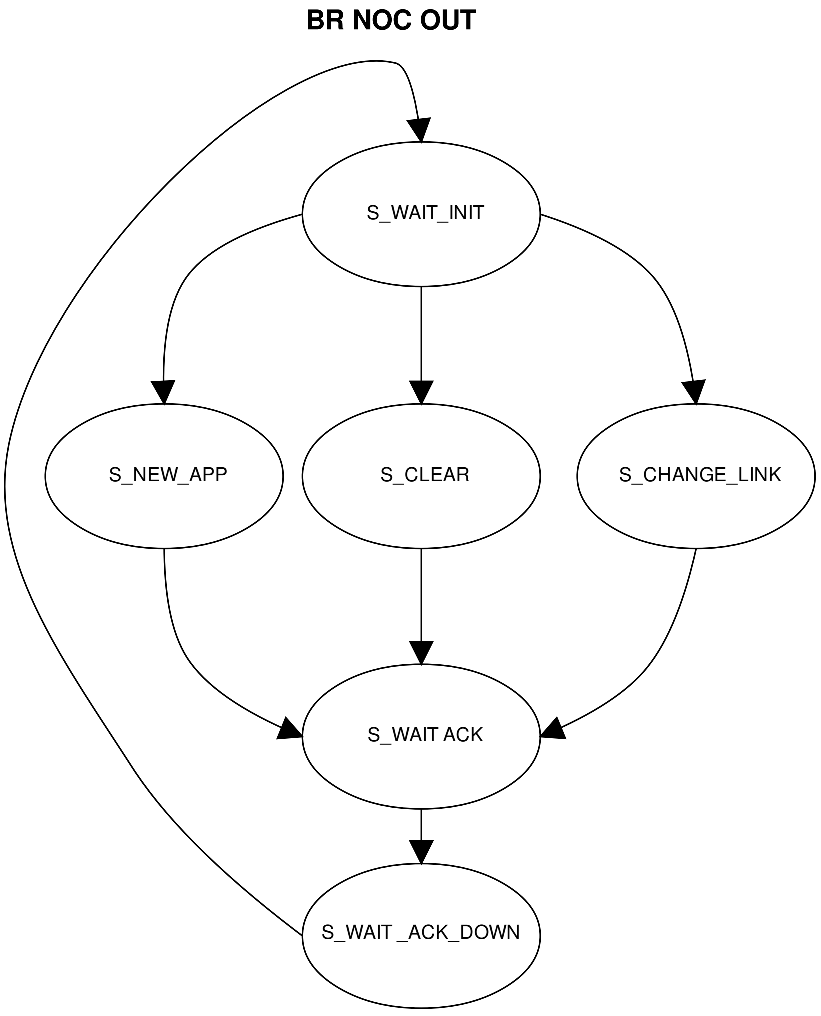


Figure A.4 – BrNoC Out NoC FSM.

BrNoC in FSM. This FSM receives packets from the control NoC of the MCSoC.

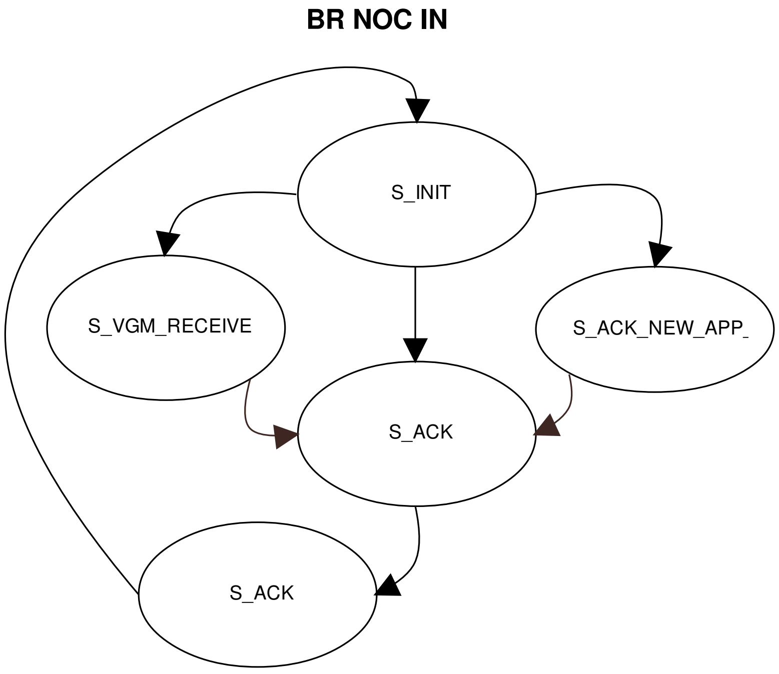


Figure A.5 – BrNoC in NoC FSM.

MARISTA

PUCRS

AD VERVM DVCIT

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br