Introduction of Fault-Tolerance Mechanisms for Permanent Failures in Coherent Shared-Memory Many-Core Architectures

César Fuguet Tortolero

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Specialité

Informatique

École Doctorale Informatique, Télécommunication et Électronique (Paris)

Presentée par

César Armando FUGUET TORTOLERO

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Introduction de mécanismes de tolérance aux pannes franches dans les architectures de processeur « many-core » à mémoire partagée cohérente

soutenue le 25 November, 2015
devant le jury composé de :

M. Fabien CLERMIDY	Examinateur	CEA (Grenoble)
M. Philippe COUSSY	Rapporteur	Lab-STICC (Lorient)
Mme. Agnès FRITSCH	Examinatrice	Thales Communications & Security (Paris)
M. Alain GREINER	Directeur de thèse	LIP6 (Paris)
M. Lionel LACASSAGNE	Examinateur	LIP6 (Paris)
M. Gilles SASSATELLI	Rapporteur	LIRMM (Montpellier)

Dissertation submitted to
THE UNIVERSITY PIERRE ET MARIE CURIE

Department of

Computer Sciences

École Doctorale Informatique, Télécommunication et Électronique (Paris)

Presented by

César Armando FUGUET TORTOLERO

For the degree of

DOCTOR OF PHYLOSOPHY

Subject:

Introduction of Fault Tolerance Mechanisms for Permanent Failures in Coherent Shared-Memory Many-Core Architectures

Defense on 25 November 2015
Committee:

M. Fabien CLERMIDY	Examiner	CEA (Grenoble)
M. Philippe COUSSY	Reviewer	Lab-STICC (Lorient)
Mme. Agnès FRITSCH	Examiner	Thales Communications & Security (Paris)
M. Alain GREINER	Advisor	LIP6 (Paris)
M. Lionel LACASSAGNE	Examiner	LIP6 (Paris)
M. Gilles SASSATELLI	Reviewer	LIRMM (Montpellier)

Résumé

L'augmentation continue de la puissance de calcul requise par les applications telles que la cryptographie, la simulation, l'analyse et distribution de paquets réseau ou le traitement du signal a fait évoluer la structure interne des processeurs vers des architectures massivement parallèles (dites « many-core »). Ces architectures peuvent contenir des centaines, voire des milliers de cœurs afin de fournir une puissance de calcul importante avec une consommation énergétique raisonnable.

Néanmoins, l'importante densité de transistors fait que ces architectures sont très susceptibles aux pannes matérielles. L'augmentation dans la variabilité du processus de fabrication, et dans les facteurs de stress des transistors, dégrade à la fois le rendement de fabrication, et leur durée de vie.

Nous proposons donc un mécanisme complet de tolérance aux pannes franches, permettant les architectures « many-core » à mémoire partagée cohérente de fonctionner dans un mode dégradé. Ce mécanisme permet à ces architectures de s'auto-reconfigurer afin de supporter aussi bien des défauts de fabrication, que des pannes de vieillissement après que la puce est en service dans l'équipement.

Le mécanisme s'appuie sur un logiciel embarqué et distribué dans des mémoires sur puce (« firmware »), qui est exécuté par les cœurs à chaque démarrage du processeur, sans aucune intervention externe. Ce logiciel implémente plusieurs algorithmes distribués permettant de localiser les composants défaillants (cœurs, bancs mémoires, et routeurs des réseaux sur puce), de reconfigurer l'architecture matérielle, et de fournir une description complète de l'infrastructure matérielle fonctionnelle au système d'exploitation.

Notre proposition est évaluée en utilisant un prototype virtuel précis au cycle d'une architecture « many-core » existante. Nous évaluons à la fois sa latence, et son coût en surface de silicium.

Remerciements

Je remercie M. Alain Greiner, professeur à l'Université Pierre et Marie Curie (UPMC) et membre du Laboratoire d'Informatique de Paris 6 (LIP6), pour avoir dirigé mes travaux de recherche. Je vous remercie d'abord pour m'avoir accueilli au laboratoire en 2011 pour faire un stage, et puis pour m'avoir proposé ce sujet de thèse, intéressant et passionnant. Merci pour votre encadrement tout au long de ces derniers quatre ans, ce fût un grand plaisir de travailler, et surtout d'apprendre à vos côtés.

J'exprime mes remerciements aux membres du jury. Je remercie M. Philippe Coussy, professeur au laboratoire Lab-STICC (Université de Bretagne-Sud), et M. Gilles Sassatelli, directeur de recherche au LIRMM (CNRS, Université de Montpellier 2), pour avoir accepté d'être rapporteurs de mon travail, et pour ses remarques très constructives. Je remercie M. Lionel Lacassagne, professeur au laboratoire LIP6, d'avoir participé et d'être président de mon jury de thèse. Je remercie également M. Fabien Clermidy, chef du laboratoire LISAN (CEA-LETI), et Mme. Agnès Fritsch, chef du laboratoire des architectures avancées à Thales, d'avoir participé à mon jury.

Je tiens à remercier M. Pirouz Bazargan, maître de conférence au LIP6, et M. Jean-Luc Danger, directeur d'études à Telecom ParisTech, pour avoir participé à mon jury de mi-parcours, et pour les remarques très constructives à mon travail.

Je remercie aux membres du laboratoire LIP6 pour créer un environnement de travail agréable et amical. Je remercie tout particulièrement à Abdelmalek Si-Merabet, Franck Wajsbürt, Pirouz Bazargan, Quentin Meunier et Manuel Bouyer. Je remercie Mme. Marie-Minerve Louërat, responsable du département SoC au LIP6, pour son aide sur le plan administratif, et je tiens aussi à remercier Mme. Shahin Mahmoodian pour sa disponibilité au secrétariat.

Je remercie pour leur amitié à tous mes collègues doctorants et ingénieurs : Clément D., Hao L., Mohammed K., Joël P., Cédric B., Benoît V., Alexandre B., Laurent L., Jean-Baptiste B. et Vanessa T.. Je remercie Kamel Hacene, que j'ai eu le plaisir d'encadrer pendant son stage sur la tolérance aux pannes dans les NoCs 3D.

Je remercie M. Gérard Paez-Monzon, professeur à l'Université des Andes (Venezuela) et ancien doctorant au LIP6, pour m'avoir introduit dans le monde de la conception des circuits numériques.

Finalement, j'étends mes remerciements à ma famille. Je remercie à ma mère et mon père pour faire de moi la personne que je suis, pour leur amour, et pour m'avoir appris à atteindre mes objectifs avec travail et patience. Merci à ma sœur Maria Alejandra, et mes neveux Antonio, Maria, et Andrés qui me donnent toujours le sourire. Je remercie ma compagne Liliana pour son amour, ses encouragements et son soutien tous les jours.

Contents

List of Figures

List of Algorithms

List of Tables

Outline

The always increasing performance demands of applications such as cryptography, scientific simulation, network packets dispatching, signal processing or even general-purpose computing has made of many-core architectures a necessary trend in the processor design. These architectures can have hundreds or thousands of processor cores, so as to provide important computational throughputs with a reasonable power consumption.

However, their important transistor density makes many-core architectures more prone to hardware failures. There is an augmentation in the fabrication process variability, and in the stress factors of transistors, which impacts both the manufacturing yield and lifetime. A potential solution to this problem is the introduction of fault-tolerance mechanisms allowing the processor to function in a degraded mode despite the presence of defective internal components.

We propose a complete in-the-field reconfiguration-based permanent failure recovery mechanism for shared-memory many-core processors. This mechanism is based on a firmware (stored in distributed on-chip read-only memories) executed at each hardware reset by the internal processor cores without any external intervention. It consists in distributed software procedures, which locate the faulty components (cores, memory banks, and network-on-chip routers), reconfigure the hardware architecture, and provide a description of the functional hardware infrastructure to the operating system.

Our proposal is evaluated using a cycle-accurate SystemC virtual prototype of an existing many-core architecture. We evaluate both its latency, and its silicon cost.

Detailed Content

- *Chapter 1-Problem Definition*, presents the motivation of this work, the context of the treated problem (many-cores, networks-on-chip, fault-tolerance, etc.), and the questions to which this work intends to answer. Additionally, this chapter describes the TSAR architecture, which is used to demonstrate our fault-tolerance mechanism.
- *Chapter 2-State of the Art*, analyzes the state-of-the-art research in the field of permanent failures recovery in many-core architectures. It presents some solutions dealing with faulty cores, or faulty NoC routers. This chapter is concluded with the problems that in our knowledge are not yet solved.
- *Chapter 3-Distributed Recovery Firmware*, presents the general principles of the proposed in-the-field software-based fault-tolerance mechanism. It gives a brief description of the software-based procedures for the location of faults, and the reconfiguration of the hardware; and introduces other contributions of this work, like the reallocation of a physical memory segment when there is a deactivated cluster, the support of broadcast communications even when the NoC is partially defective, or a reconfigurable fault-tolerant algorithm for 3D NoCs.
- *Chapter 4-Distributed Fault-Location*, presents a detailed description of the proposed software-based fault-location procedure. This chapter describes the different distributed procedures of the recovery firmware, defines the properties that these procedures must satisfy, and proves these properties. Additionally, it presents a software, but hardware-assisted mechanism to test the coherence NoCs in the TSAR architecture.
- *Chapter 5-NoC Reconfiguration*, presents the software-based distributed procedure to reconfigure the NoCs. Additionally, it presents the mechanism allowing to reallocate the physical memory segment of a deactivated cluster to one of its neighbors, and a fault-tolerant routing algorithm supporting broadcast communications.
- *Chapter 6-Experimental Results and Evaluation*, presents the evaluation in terms of latency, and silicon cost of the proposed fault-tolerance mechanism, that has been implemented in a cycle-accurate virtual prototype of the TSAR architecture.
- *Chapter 7-Fault-Tolerance Extension for Interconnects above the Computational Layer*, presents an extension to the fault-tolerance mechanism to support permanent failures in 3D NoCs. This chapter describes a software-based fault-location, and fault-reconfiguration procedure for these NoCs; and a 3D fault-tolerant routing algorithm.

Chapter 1

Problem Definition

Contents

1.1 Motivation

Many-core architectures are a current trend in processor design to face the augmentation on the performance demands of modern systems [1]. These architectures take benefit of the increasing transistor density of circuits by the integration of hundreds or thousands of small cores in a single chip. However, as a consequence of the very high transistor density, the reliability in this kind of architecture is a major concern. The technology scaling improves the transistor density but arises the two following consequences: (1) augmentation in the process variability that impacts the manufacturing yield and (2) augmentation of the stress factors of transistors that impacts the lifetime of Integrated Circuits (ICs) [2].

The International Technology Roadmap for Semiconductors (ITRS) [3] estimates that in the near-future, "the cost of ensuring that each transistor in a large integrated circuit to function within specification may become to high to be practical" and therefore, there will be a high percentage of non-functional devices right after fabrication. In the ITRS design report at 2011, the Figure 1.1 shows failure probabilities predictions (y-axis) against the technology nodes (x-axis) for two major components in the digital CMOS design: SRAM bitcell and latch. The figure shows that failure probabilities can reach 10% for SRAM cells in the 16nm technology if traditional circuit design approaches continue unchanged; therefore shows the necessity of introducing new circuit and architecture techniques.

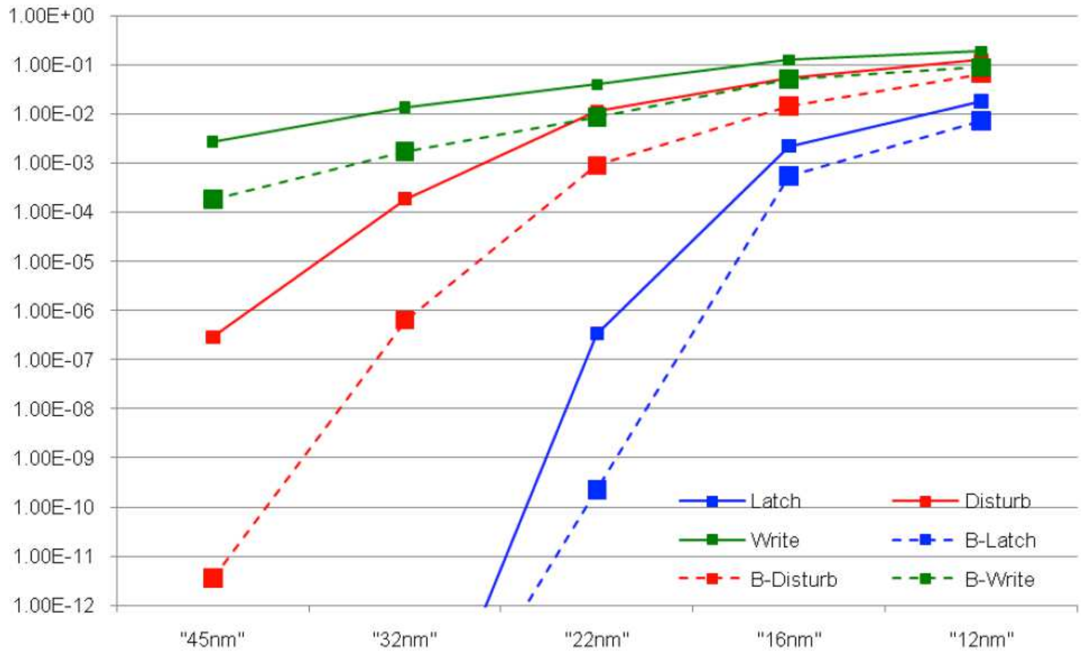


Figure 1.1 – Variability-Induced Failure Rates for Two Canonical Circuit Types (source: ITRS [3])

A potential solution to this massively defective circuit problem can be the introduction of fault-tolerance mechanisms. These mechanisms improve both yield and lifetime factors by allowing hardware to function, in a possibly degraded mode, despite the presence of defective internal components. Such a solution is achieved by designing systems that can dynamically self-reconfigure to deactivate the internal failing devices. In order to enable self-reconfiguration during the entire life-time of the circuit, fault-tolerance should be achieved by means of on-chip components without any external intervention. Such kind of solutions are denominated “in-the-field”. These solutions improve the manufacturing yield by allowing to produce circuits with a tolerated percentage of faulty internal devices and also to improve the circuit’s lifetime by allowing it to reconfigure when it is already in service after manufacturing (in the field).

1.2 Many-core Architectures

Many-core architectures exploit thread level parallelism through hundreds or thousands of cores and therefore, applications should exhibit highly parallel behaviors. Some examples of applications are cryptography, finance, weather forecasting, scientific simulation, network packets acquisition and dispatching, or signal processing (e.g. video compression). They can also be used in embedded environments for automotive or aerospace applications.

In order to reduce architecture bottlenecks, and therefore improve the parallel execution, many-cores are usually organized in clusters, interconnected through a Network-on-Chip (NoC), where each cluster contains various components such as one or more cores, local memory banks and internal peripherals. This organization allows a scalable performance by providing important memory and communication bandwidths.

The communication between cores can be implemented using whether a shared memory or a message-passing paradigm. On the former, all cores share the same address space, thus any processor core can write or read any memory location (local or remote). On the latter, all processor cores in a cluster have a private address space, and hence software needs to explicitly move data from the local memory bank to the remote memory bank by means of message passing in order to perform communication between clusters. This work is interested in many-core architectures implementing a shared memory address space.

Additionally, to reduce access time and to decrease demands on external memories, processors implement a memory hierarchy containing one or more levels of cache memory (lower levels) followed by external main memory and disk (higher levels). When one or more cache levels are private to a processor core in a shared memory architecture, a cache coherence problem exists. As a data replica can be at several private caches, modifications on a replica should be propagated to all other replicas in order to have a consistent view of the memory by all processor cores. The propagation of changes on replicas could be done by software or hardware. This work is interested in architectures guarantying cache coherence by hardware.

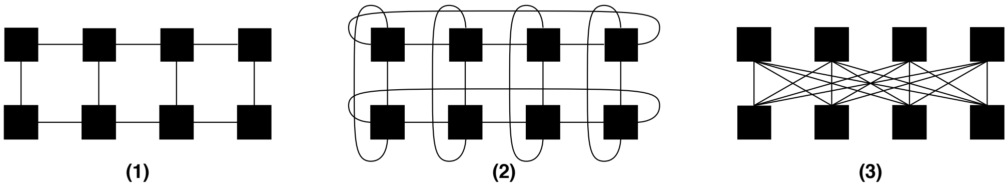


Figure 1.2 – Examples of NoC Topologies: (1) 2D Mesh, (2) 2D Torus, (3) 4 × 4 Crossbar

1.3 Network-on-Chip (NoC)

NoCs are in charge of the interconnection of internal devices in many-core architectures. Basic components of NoCs are routers and communication channels: routers are in charge of the routing of incoming packets from one input channel to one output channel; and channels are buffering devices connecting one output port of a router to an input port of another router.

In general, to avoid communication bottlenecks, the NoCs use regular topologies providing a scalable communication bandwidth. When the number of devices is small, they can be interconnected using a fully connected topology like a crossbar (see Figure 1.2 (3)). Otherwise, if the number of devices is important, the NoCs use partially connected topologies like multidimensional meshes. Figure 1.2 (1) and Figure 1.2 (2) show a Two-Dimensional (2D) mesh and torus respectively. In this work, we consider many-core architectures implementing multidimensional meshes for the inter-cluster connections as they are the most commonly used. Such topologies allow, on the one hand to reduce the silicon cost by the implementation of simple routing algorithms, and on the other hand to reduce the use of long wires that incur important delays and power dissipation [4].

1.3.1 Globally-Asynchronous Locally-Synchronous (GALS)

One important feature proposed by modern NoCs is the implementation of the Globally-Asynchronous Locally-Synchronous (GALS) approach. As the size of the systems grows, is becoming harder to provide a single clock through the whole chip because of the skew. The skew is produced among other reasons, by the delay incurred by wires while sending the clock signal to the different sequential devices of the chip. When the clock signal has significantly different arrival times to the different devices, timing violations may occur. Moreover, with the increment in the clock frequency such violations are becoming more important.

The term GALS was firstly used by Chapiro [5] and it is a technique consisting in dividing the chip in several clock-independent regions (clock domains). Each region is locally synchronous but may be asynchronous with respect to other regions. In order to allow communication between clock domains, the regions can use bi-synchronous FIFOs at their interfaces. The advantages of this technique are: modularity, scalability, smaller clock wires length because there is no need of circuit's global clock, power consumption improvements and fault-tolerance.

The power consumption can be improved in two ways: 1) smaller clock wire lengths results in smaller wire resistance and decreases power consumption, and 2) as each region can work with different clock frequencies, one can for example dynamically slow down the clock of currently unused regions or regions executing some low priority task. Regarding fault-tolerance, because there are several clocks, the failure of one of them only affects the concerned region.

1.3.2 Routing Algorithm

The routing algorithm defines which network path is used by packets in order to go from a source cluster to a destination cluster.

When the path is always the same for all packets transferred between a given pair of clusters, the routing algorithm is deterministic, and the path depends only on the packet's destination. Otherwise, when the path depends on the destination and on the link status (e.g. network congestion), the routing algorithm is adaptive. The adaptive routing algorithms use network resources better but they are more expensive than deterministic counterparts. In particular, adaptive solutions do not guarantee the in-order delivery property and therefore, need specific hardware in the Network Interface Controllers (NICs) to reorder packets. This kind of mechanism is then not always scalable with respect to area, energy consumption and latency. Therefore, deterministic routing algorithms are commonly preferred in NoCs.

The routing algorithm can be implemented using whether routing tables (a.k.a. forwarding tables) or a logic-based circuit. When using the former, there is one table per router that contains at least one entry per destination (per node in the network). Each entry contains the output port associated to a destination. When using a deterministic algorithm, there is exactly one entry per destination. In the case of logic-based routing algorithms, the output port is computed by a hardwired combinational circuit that takes as an input the destination of the packet. Routing tables are more flexible concerning topologies but they are not scalable because the number of entries depends on the number of nodes in the network. As a consequence, the logic-based implementations are preferred for NoCs because of their scalability as the area does not depend on the network size.

An important property that routing algorithm must guarantee is that a packet always reaches its destination. Therefore, it must guard against livelock and deadlock situations. The former arises when a packet can be routed an unbounded time in the network. The latter arises when a packet cannot advance towards its destination because it waits for a network resource to be freed by another packet. Both problems can be solved by implementing some restrictions on the routing algorithm. In the case of livelocks, the number of forwarding options of routers should be restricted to avoid the packet to be redirected indefinitely without reaching its destination. When using deterministic routing algorithms there is a unique possible path between any pair of clusters and therefore there is no risk of livelock. In the case of deadlock situations, they can be avoided by eliminating cycles in the dependency graph of the network resources [6].

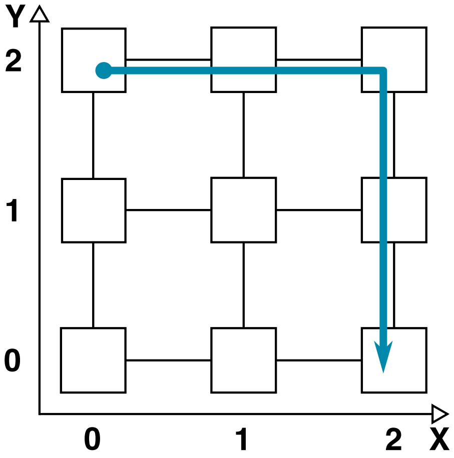


Figure 1.3 – X-First Routing Example

Usually used livelock-free, deadlock-free, deterministic and logic-based routing algorithms are the Dimension-Order Routing (DOR) algorithms. These algorithms work on multidimensional meshes or torus. The packets are routed in a specified order of dimensions until it reaches its destination. Figure 1.3 shows an example for a 2D mesh: a packet is first routed on the X dimension and then on the Y dimension. This specific case of DOR algorithms is called X-first routing (a.k.a. XY routing).

DOR algorithms have the advantage to be simple, and therefore cost-effective solutions for NoCs. However, they do not support irregular topologies. When some NoC routers or channels are faulty in a regular topology, it may become irregular, and then DOR algorithms cannot support it. In such cases, the communication between clusters is broken, and the entire chip is useless. This is why some modifications need to be introduced in such algorithms in order to support some irregular topologies while preserving their low-cost, deadlock free and livelock free properties. Such modifications will be presented in Section 1.5.

1.4 Tera-Scale Architecture (TSAR)

The architecture used by this work to validate and evaluate the proposed fault tolerance technique is TSAR.

TSAR [7] is a scalable, cache-coherent, shared-memory many-core jointly designed by BULL, LIP6 and CEA-LETI in the framework of the European CATRENE SHARP project. It supports up to 1024 processor cores organized as a mesh of clusters, where each cluster contains up to 4 processor cores as shown in Figure 1.4. Each processor core has a private L1 data cache and L1 instruction cache. Additionally, each cluster contains a shared L2 cache controller, which can be accessed by any processor core in the system, an interrupt controller (XICU) and a Direct Access Memory (DMA) controller.

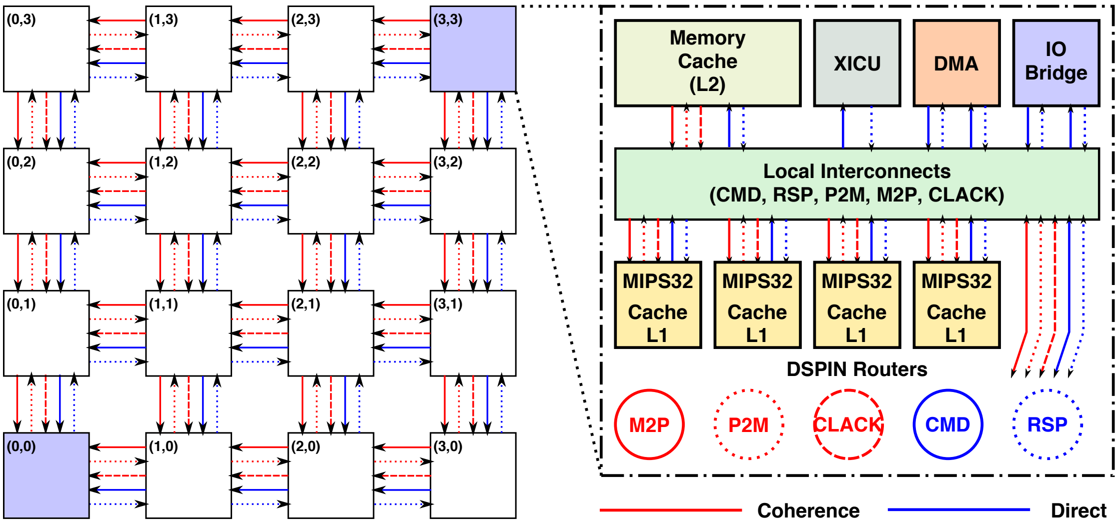


Figure 1.4 – TSAR 2D-Mesh

1.4.1 Memory Hierarchy

In Tera-Scale Architecture (TSAR) the memory is logically shared but physically distributed. This means that every memory location is shared and can be accessed by any processor core, but the address space is physically distributed in every cluster. Each cluster manages an exclusive segment of the physical address space defined by the most significant bits of the physical address. As a result of this address space distribution, the TSAR is a Non-Uniform Memory Access (NUMA) architecture, i.e., the memory access latency is not uniform for the entire address space but depends on the distance between the processor core and the target memory location.

The physical address space length is 1 Terabyte (40 bits addresses) but to be energy efficient, this architecture uses 32-bits, single instruction issue, RISC cores with a 4 Gigabytes address space (32-bits addresses). These cores implement the MIPS32 Instruction Set Architecture (ISA) with a Memory Management Unit (MMU) containing two separates Translation Lookaside Buffers (TLBs): one for data and one for instructions. This custom MMU implementation allows, in particular, to maintain the TLBs coherence by hardware.

In addition to the L1 and L2 cache controllers, TSAR implements a third level of cache (L3). Nominal capacities of L2 and L3 caches are 256 Kbytes and 1 Mbyte, respectively. The L1 data and instruction caches have each 16 Kbytes. The Figure 1.5 contains a flattened global view of the TSAR architecture with its memory hierarchy.

Regarding cache coherence, because L1 caches are private but L2 caches are shared, TSAR implements a hardware cache-coherence protocol called Distributed Hybrid Cache Coherence Protocol (DHCCP). The L1 caches implements a *write-through* policy while the L2 caches implements a *write-back* policy.

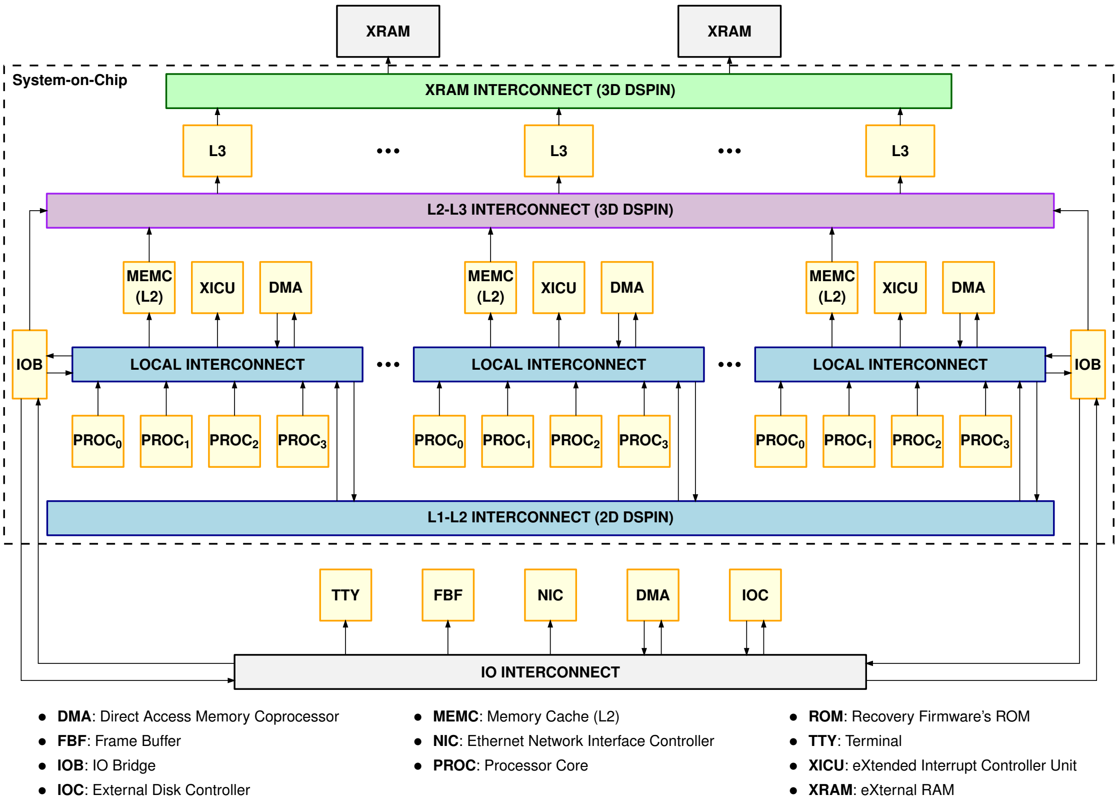

XRAM
XRAM
System-on-Chip
XRAM INTERCONNECT (3D DSPIN)
L3
L3
L3
L2-L3 INTERCONNECT (3D DSPIN)
MEMC (L2)
XICU
DMA
IOB
LOCAL INTERCONNECT
PROC0
PROC1
PROC2
PROC3
L1-L2 INTERCONNECT (2D DSPIN)
TTY
FBF
NIC
DMA
IOC
IO INTERCONNECT
DMA: Direct Access Memory Coprocessor
FBF: Frame Buffer
IOB: IO Bridge
IOC: External Disk Controller
MEMC: Memory Cache (L2)
NIC: Ethernet Network Interface Controller
PROC: Processor Core
ROM: Recovery Firmware's ROM
TTY: Terminal
XICU: eXtended Interrupt Controller Unit
XRAM: eXternal RAM

Figure 1.5 – TSAR Flattened Global View

1.4.2 Networks-on-Chip

Several independent internal NoCs are implemented in TSAR. The Figure 1.6 shows the physical implementation. It relies on a Three-Dimensional (3D) stacking technology: in the first layer (called computational layer) are the cores, L1 cache and L2 caches. The other layers contain the distributed L3 cache.

The computational layer of TSAR (shown in Figure 1.4) implements five physically independent NoCs (CMD, RSP, M2P, P2M & CLACK): two for the direct traffic (normal read and write transactions triggered by L1 cache controllers or DMA-capable internal peripherals) and three for the coherence traffic (triggered by L1 and L2 cache controllers to maintain the memory coherency). The CMD and RSP NoCs transport the commands and responses of the direct transactions, respectively. The M2P and CLACK NoCs transport commands from the L2 cache to L1 cache controllers and the P2M NoC transport commands from the L1 cache to L2 cache controllers. These three independent NoCs are required by the coherence protocol because some transactions need three phases. Thus, for deadlock avoiding, the messages on each phase must use independent networks.

The five NoCs in the computational layer implement a two-level hierarchical structure: local and global. The local interconnect uses a crossbar for intracluster communication, and the global interconnect uses the 2D-mesh Distributed, Scalable, Programmable, Integrated Network (DSPIN) for intercluster communication.

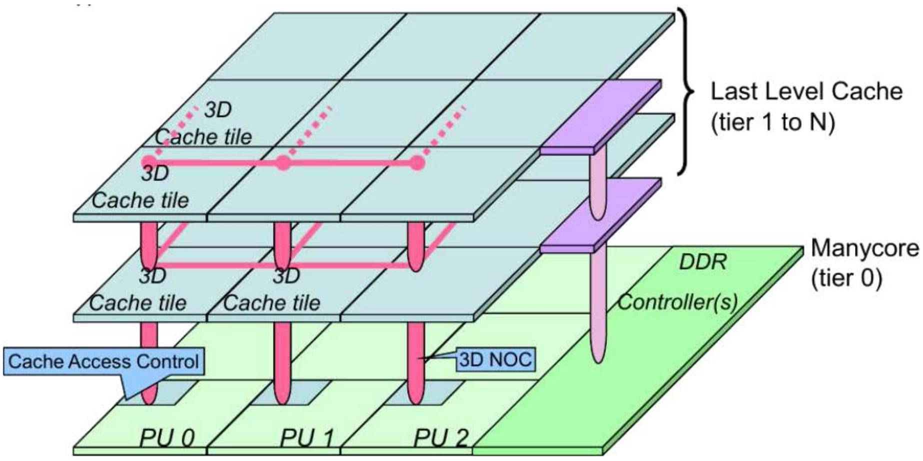


Figure 1.6 – TSAR 3D Stacking Technology (source: Guthmuller, Miro-Panades, and Greiner [8])

Miro Panades, Greiner, and Sheibanyrad [9] define the DSPIN NoC infrastructure which is intended for clustered shared memory multiprocessor architectures. It implements a 2D mesh topology and uses the deterministic, deadlock and livelock free, in-order delivery, X-first routing algorithm. Additionally, it implements the GALS approach, i.e., each cluster in the architecture can have its own clock source with a possibly different frequency with respect to other clusters. The communication between clock domains is achieved through bi-synchronous FIFO buffers in the router-to-router interconnection channels.

Guthmuller, Miro-Panades, and Greiner [8] define the L3 cache architecture implemented in TSAR. The L3 cache layers use a 3D version of the DSPIN infrastructure for NoCs. There are two independent 3D NoCs for L2 and L3 cache controllers command and response traffics. Additionally, there are two 3D NoCs for command and response traffics between L3 cache controllers and the external memory banks (XRAM). Table 1.1 shows the routing algorithms used by these NoCs. In summary, TSAR implements nine internal NoCs.

NoC	Routing Algorithm
L2 → L3	*ZXY*
L3 → L2	*XYThenDown*
L3 → XRAM	*XYThenDown*
XRAM → L3	*ZXY*

Table 1.1 – Routing Algorithms Used on TSAR 3D NoCs

1.4.3 IO Subsystem

Input/Output (IO) peripherals are connected to an external IO interconnect. TSAR implements bridges placed in specific clusters of the computational layer to access this interconnect.

As shown in Figure 1.4, TSAR implements two IO bridges: one on cluster (*0*, *0*) and another on cluster (*XSIZE* − *1*, *YSIZE* − *1*) where *XSIZE* and *YSIZE* are the size of the 2D mesh on *X* and *Y* dimensions, respectively. These two clusters are called in the remainder of this document as IO clusters. The two IO bridges are connected to the same IO interconnect and therefore, provide two different paths to access external peripherals. This redundancy is important for fault-tolerance purposes: if one IO cluster is faulty, the other one can be used.

1.5 Fault-Tolerance

Fault tolerant systems are capable of operating despite the presence of faults. For this purpose, systems must implement fault tolerance techniques whose complexity depends on the reliability requirements. One of the requirements that must be defined is the kind of tolerated faults.

When classifying faults by duration, two coarse categories can be distinguished: permanent faults and transient faults. Permanent faults remain in existence indefinitely and transient faults appear and disappear within short periods of time. The formers can appear as the consequence of manufacturing defects or aging; the latter can appear because of external disturbances effects like radiation. In modern ICs the problem of permanent failures is becoming important because of the process variability, and the accelerated aging issues of current technology nodes. Such issues are a consequence of the technology scaling and the related very high transistor density. This work focusses in fault tolerance techniques for permanent failures.

There are several types of fault tolerance techniques. The dynamic reconfiguration approach tries to recover a desirable state of the system when faults have been detected. The reconfiguration can consist in replacing faulty components by backup spares, or isolating faulty components from the rest of the system. Techniques that isolate and deactivate faulty components, are called graceful degradation techniques. Such techniques use the redundant hardware resources of the system to allow a degraded operational mode. This work focusses on this kind of techniques.

Regarding many-core architectures, graceful degradation techniques are an interesting fault tolerance approach because these architectures are inherently redundant (several identical cores, memory banks and NoC routers). Therefore, one can imagine various operational modes where faulty cores, memory banks or NoC resources are deactivated and the remainder of the system continues to perform its functions with only a limited performance degradation.

The system reconfiguration can be performed by means of external or internal devices (or both). When the reconfiguration is performed autonomously by internal devices, the system is called in-the-field self-reconfigurable (a.k.a. self-healing systems or self-adaptable systems). Such systems are able to *heal* themselves after the occurrences of failures on internal devices during its entire lifetime. Therefore, they can self-reconfigure to recover from manufacturing defects or from components wear-out. This work searches an in-the-field self-reconfiguration solution that improves both manufacturing yield and lifetime of many-core processor chips.

The reconfiguration-based fault tolerances approaches involve four issues [10]: fault detection, fault location, fault containment, and fault recovery. Fault detection is the ability of a system to recognize that a fault occurred; fault location is the process of determining where a fault has occurred; fault containment is the process of isolating a fault and preventing its effects from propagating throughout the system; and fault recovery is the ability of the system to regain an operational mode via reconfiguration.

Fault Detection

The first process performed in any reconfiguration-based fault tolerance technique is the fault detection. In this process the erroneous behavior of internal components is detected, in order to allow a subsequent location and deactivation of failed components.

This work tries to reuse existent fault detection techniques and therefore, it focusses on how to solve the fault location, fault containment and fault recovery issues. Hereafter a brief introduction of some existent fault detection techniques is presented.

In general, the fault detection process consists on applying test patterns to the inputs of the Device Under Test (DUT), and the outputs are compared to expected values. Errors are detected when there are differences between the obtained values and the expected ones. For the purpose of applying test patterns, there are several techniques and among these techniques there are: Automated Test Equipment (ATE), Built-In Self-Test (BIST) and Software-Based Self-Test (SBST).

The ATE is an external test equipment that is connected to inputs and outputs of the entire chip. It performs comparisons against expected values in order to detect errors. This approach is not suited for in-the-field fault detection.

The BIST techniques were introduced to perform test at the system nominal clock frequency and allow the DUT to self-diagnose. In this technique, dedicated test pattern generators and analyzers are introduced in the chip. However, when the complexity of the DUT is important, the BIST overhead may also be important.

Finally, the SBST solution uses the internal processor cores to run test procedures that are generally stored either in internal or external Read-Only Memory (ROM). These test procedures allow the core to diagnose itself and the memory subsystem. As the core uses its native instruction set to generate and analyze test patterns, there is no need of additional test-specific hardware. Test procedures write and read different memory locations, and perform comparisons to detect erroneous behaviors. Like BISTs, the SBSTs can be run in the field at the nominal clock frequency of the system.

In this work we plan to use pre-existent BIST solutions for the NoCs, and pre-existent SBSTs solutions for the processor cores and memories. NoC routers have a rather low hardware complexity, and the associated overhead of the BIST mechanism is not important.

In order to implement the fault detection on NoCs, this work intends to use a distributed BIST like the one defined by Zhang, Greiner, and Benabdenbi [11]. This BIST is distributed in every router, and it performs the router deactivation when a fault is detected at every processor power-on.

Regarding the processor cores, several SBST solutions exist [12]–[14]. These solutions allow the test of a core and the memory subsystem, by means of self-testing programs. As reference, in Kranitis, Paschalis, Gizopoulos, *et al.* [12], the authors propose a SBST for a MIPS32 processor with a 5-stage pipeline. This SBST achieves a 92% fault coverage, with a latency of 10 Kcycles, and 7 Kbytes of code.

1.5.1 NoCs Routing Algorithm Reconfiguration

The use of regular topologies in the NoCs, provides an intrinsic redundancy which can be used for fault-tolerance. In fact, in this kind of topology there are several paths between any pair of communicating nodes. However, in order to reduce the silicon cost and guaranty in-order delivery, the NoCs use deterministic routing algorithms where any pair of nodes communicate to each other through a deterministic path. In this scenario, when there is a faulty router in a path, the communication is broken. Therefore, it is necessary to introduce a reconfiguration mechanism in the routers in order to modify their routing algorithm and then, reestablish communication by bypassing the faulty components. However, this reconfiguration must guaranty that the new routing algorithm is deadlock and livelock free.

The solution proposed by Zhang, Greiner, and Taktak [15] deals with the reconfiguration of the 2D DSPIN interconnect in order to bypass a faulty router. This solution consists in the introduction of a configuration register in the DSPIN routers that allows modifying the default X-first routing. However, this solution needs that the faulty routers (creating *holes* in the network) are precisely located, and it requires a reconfiguration communication infrastructure to reliably reconfigure the routers. This work searches solutions to these two problems.

1.5.2 Distributed Algorithms

The location of faulty routers is required to reconfigure the remaining routers and adapt to the new irregular topology. The main problem is to centralize all this distributed information (location of faulty routers) to compute a new global routing algorithm.

This work will search a solution to locate faulty routers by executing a distributed firmware. The goal is to reuse the processor cores to reduce the overall hardware cost of the solution. However, this implies that the fault tolerance mechanism is running on an unreliable hardware as some processor cores, memory banks or communication links can be faulty.

For this reason, the solution to this problem requires distributed algorithms. Such algorithms are executed in parallel by processor cores, but they need to work coordinately in order to centralize the distributed gathered information and take consensual decisions. Moreover, these algorithms must consider the problem of being executed on unreliable hardware and this is one of the main challenges that this work has to face.

1.6 Problem Definition

The technology scaling allows circuits to improve their computational power by increasing the transistor density. However, there is also an increase of the probability of failures, and the cost of ensuring that every transistor in the chip works within the specifications is becoming not practical. A possible approach to deal with these defective chips is to introduce reconfiguration-based fault-tolerant mechanisms. Such mechanisms can improve both the manufacturing yield and lifetime by allowing the circuits to work degradedly despite the presence of faults.

Many-core architectures are inherently redundant. They contain multiple cores and memory banks. The regular NoC topologies provide redundant communication paths between internal components. Therefore, the general question to which this work will intend to answer is: **how to take benefit of the many-core architectures' intrinsic redundancy in order to tolerate permanent failures in cores, memory banks and NoC components?**. This work considers in-the-field reconfiguration (i.e. after the chip manufacturing) without any external intervention. Besides, in order to reduce the overhead of the fault-tolerance mechanism, this work focusses in solutions reusing at maximum the existent hardware for the reconfiguration (e.g. reuse of internal cores, memories and communication infrastructure). Such a solution can be based on distributed software procedures executed in parallel by all processor cores, on the one hand, to provide scalable reconfiguration times and, on the other hand, to provide robustness because the reconfiguration does not depend on a unique non-faulty device.

Regarding the fault-tolerance on the NoCs, our work relies upon two existent technologies: the first, proposed by Zhang, Greiner, and Benabdenbi [11] provides a BIST for the detection and deactivation at every hardware reset of the faulty routers in a 2D mesh NoC; the second, proposed by Zhang, Greiner, and Taktak [15] provides a reconfigurable dead-lock free routing function for this kind of NoC which allows bypassing the deactivated faulty routers. However, in order to perform the reconfiguration of the NoC, we need to locate the deactivated routers, compute the new global routing algorithm, and provide a reliable communication infrastructure to reconfigure the routers.

Thus, this work intends to answer the two following questions: (1) **how to locate the faulty routers, faulty cores, and faulty memory banks in order to build a map of the operational hardware devices ?** and (2) **how to reconfigure the routers reliably when the existent hardware communication infrastructure is partially defective?**. As explained in Section 1.1, many-core architectures have several internal NoCs allowing the transmission of different types of messages (e.g. normal memory accesses, hardware cache coherence) between different internal devices. This work considers the reconfiguration of all internal NoCs to effectively increase the fault tolerance in many-cores.

Additionally, we focus in many-core processors capable of executing commodity Operating Systems (OSs) like Linux or NetBSD. Such OSs need to determine the complete description of the hardware in order to initialize their internal data structures. In the presence of hard faults, the available devices may change and thus, the OS needs to know about these changes. Our work answers the following question: **how to transmit the map of the functional hardware devices to the OS so as to support its execution on a degraded architecture?**.

Chapter 2

State of the Art

Contents

This chapter analyses the state-of-the-art research in the field of permanent failures recovery in many-core architectures. It is divided in three sections.

The first section presents fault-tolerance techniques dealing with permanent failures in many-core processors. The second section focuses in the NoC, and presents some fault-tolerant routing algorithms to support faulty routers or links. Finally, there is a conclusion section, where we present the problems that in our consideration are not yet solved, and for which we will propose a solution.

2.1 Fault-Tolerance in Many-Core Processors

This section presents three solutions related to the problem of supporting permanent failures in many-core processors.

2.1.1 Many-Core Yield Enhancement

In Zhang, Han, Xu, *et al.* [16], the authors propose the **N+M** fault-tolerance technique for yield enhancement. This technique uses a core-level redundancy scheme where a N-core processor is fabricated with M spare cores.

The idea is to always provide **N** cores to customers, in spite of the presence of faulty cores. The chip is reconfigured after fabrication, and it can be repaired only if there are at most **M** faulty cores. This technique does not consider faults in the NoC.

This kind of solution has an important hardware overhead by introducing spare cores which are not used during normal operation. And, the cost of tolerating several faulty cores is restrictive because it depends on the number of spares. More importantly, this work does not support in-the-field reconfiguration, as the proposed technique does not provide any self-reconfiguration mechanism.

2.1.2 Many-Core Self-Organization

In Zajac and Collet [17] and Collet, Zajac, Psarakis, *et al.* [18], the authors propose a self-organization approach to support permanent failures recovery in the field. This approach considers the problem of diagnosing cores, and the problem of discovering communication routes in many-core processors at power-on.

The core diagnosis is based on a software mutual test. Each core executes a diagnosis program locally stored in a flash-like memory. A core tests itself and its direct neighbors. When a good core detects a problem on one of its neighbors, it stops all communications with this one. The NoC diagnosis uses a hardware-based method to test the point to point communication channels and routers, and is performed independently of the core diagnosis.

Regarding the communication routes discovery, all nodes broadcast a message to discover the routes to reach all the other good nodes. During this process, the routers use a specific hardware mechanism to broadcast the message packet, and add the local routing decision to the packet header. When a node receives this packet, the followed route (path) is contained in the header, and the receiving node sends an acknowledgement to the emitter which follows the same route. Finally, after the reception of acknowledgements, the emitter builds and stores an array of routes to contact other good nodes.

The routing algorithm of the NoC is not specified by the authors, but it can be understood that they use a NoC with source routing as the packages header contain the route to follow. With source routing, the emitter (source) sends in the packet header the entire path (route) to reach the destination. This kind of routing algorithms are inherently fault-tolerant because source routing supports any topology (regular or not).

However, this approach has two important overheads with respect to distributed routing algorithms where each router computes locally the forwarding link: 1) packets are bigger because they need to store the paths, and 2) the routers need a buffer to store the routes to all nodes of the mesh. Therefore, NoCs implementing source routing are not scalable. As explained in Chapter 1, distributed deterministic routing algorithms are preferred for NoCs because of their low hardware complexity. Of course, they need to be reconfigurable in order to be fault tolerant.

2.1.3 Many-Core Distributed Cores Diagnosis

In Kamran and Navabi [19], the authors propose a scalable in-the-field test architecture to distribute test stimuli among homogeneous processing cores in many-core processors.

All cores are tested by executing locally a SBST. However, instead of storing the entire SBST in each cluster, this solution proposes the use of small test buffers (about ten words) at each cluster, which are filled periodically by means of a specific hardware network. The SBST is stored in one on-chip or off-chip memory, and is divided in small pieces of code called test-snippets. The test-snippets are broadcast one by one through the specific hardware network to all test buffers, and are executed locally by cores. When the cores finish the execution of all test snippets, another hardware device centralize the test results using another specific network, and diagnoses the cores.

This solution proposes an alternative to the overhead of distributed on-chip ROMs containing SBSTs. Instead, the proposed mechanism uses a specific hardware communication network and buffers for the test distribution and result centralization. However, this makes the solution more prone to failures because a single fault in these specific networks prevents the cores' diagnosis. The advantage of using distributed on-chip ROMs, is that a faulty ROM only affects the cores in the cluster containing the faulty ROM.

Moreover, this work focuses on the test of cores. Therefore, it needs to be completed with reconfiguration-based recovery strategies allowing many-core processors to self-reconfigure, and operate despite the presence of faulty components.

2.2 Fault-Tolerant Routing Algorithms for NoCs

This section presents four solutions to support permanent failures in the NoC of many-core processors. These solutions propose fault-tolerant algorithms allowing a NoC to route packets on irregular topologies, resulting from the combination of faults in routers or point to point communication channels.

2.2.1 Fault-Tolerant Routing Based on Virtual Channels

In Chaix and Avresky [20], [21], the authors propose an adaptive deadlock-free routing algorithm using virtual channels. The proposed routing algorithm implements two virtual channels in the NoC, and these virtual channels form two virtual networks which implement each a different routing algorithm: one uses the north-last algorithm, and the other uses the south-last algorithm. Cunningham and Avresky [22] presents these routing algorithms.

Each of these routing algorithms are deadlock-free because they prohibit some turns as for the X-first routing algorithm presented in Chapter 1. However, there are less prohibited turns to provide adaptability. Therefore, there can exist several communication paths between two nodes, and the routers make dynamic routing decisions based on the neighbors' status (faulty or functional). This adaptability allows bypassing faulty routers and links in the NoC.

This kind of adaptive routing algorithms are an interesting approach to support in-the-field fault-tolerance in NoCs. Such routing algorithms tolerate all single faulty router or link topologies, and several multiple faulty router or links topologies. However, the use of virtual channels represents an important hardware overhead. In fact, the introduction of virtual channels in routers increases significantly their area and delays because of the extra buffering, switching and arbitrating [23].

2.2.2 Segment-Based Routing Algorithm

In Mejia, Flich, Duato, *et al.* [24], the authors propose a routing algorithm supporting irregular topologies, which can result as a consequence of faults. This algorithm, called **segment-based routing (SR)**, was first proposed for parallel computers or clusters but can be applied in many-core NoCs.

The segment-based routing algorithm is deterministic and do not use virtual channels. It works by partitioning the network into segments, which are composed by routers and links. These segments are disjoint (a router or link belongs to one segment only). Within each segment, a bidirectional routing restriction is introduced to break cycles. At the end, the entire network has no cycle, and the routing algorithm is deadlock free.

This algorithm is topology agnostic, and can handle any topology derived from any combination of faults. However, it needs routing tables at each router, with one entry per possible destination, so it is not a scalable solution, as explained in Chapter 1.

2.2.3 Logic-Based Distributed Routing (LBDR)

In Rodrigo, Medardoni, Flich, *et al.* [25], the authors propose the **Logic-Based Distributed Routing (LBDR)** mechanism to implement distributed routing algorithms in 2D meshes. This mechanism replace routing tables in routers by a register with three bits per output port, and a small logic of several gates. In 2D meshes, this results in a 12-bits register per router. This LBDR mechanism allows implementing the so-called minimal adaptive routing algorithms: only shortest paths defined in the original 2D mesh (2D mesh without faults) can be chosen.

When the SR algorithm [24], presented above, is implemented with the LBDR mechanism, the resulting routing algorithm allows tolerating various irregular topologies, derived from some combinations of faults, with a small hardware overhead. Examples of supported irregular topologies are shown in Figure 2.1 [①-⑥]. However, as a consequence of the minimal path restriction, some simple faulty topologies are not supported. For example, as shown in Figure 2.1 ⑦, a single faulty router in the center of the mesh is not supported by LBDR because some packets need a non-minimal path to bypass the faulty router. In the original topology (illustrated in Figure 2.1 ①), a packet from node (2,2) to node (2,0) would take two hops. However, if the router (2,1) is faulty, the packets need at least four hops to reach its destination.

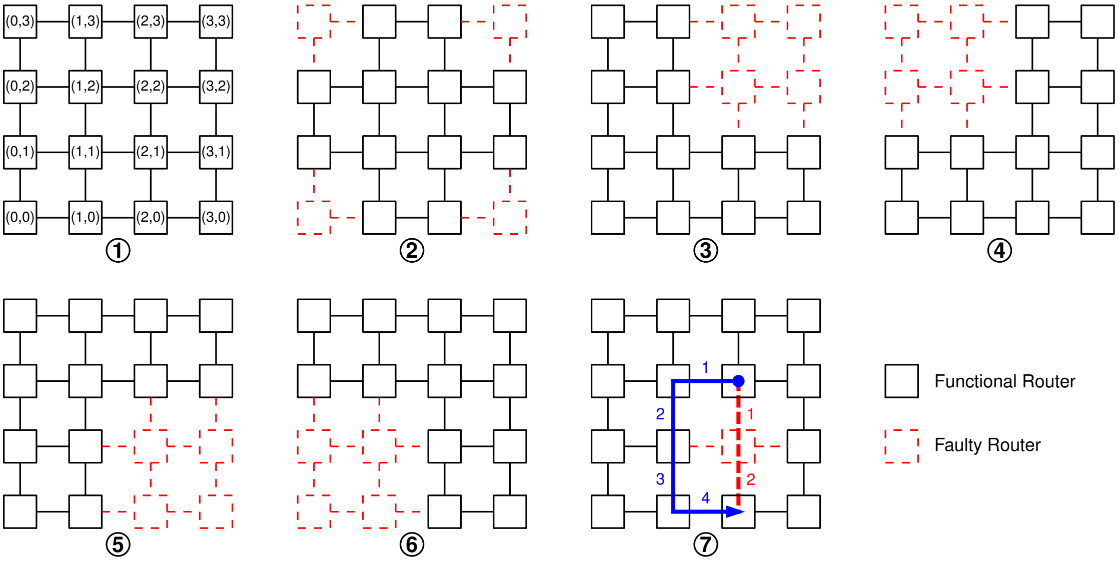


Figure 2.1 – Examples of topologies: 1,2,3,4,5,6 are supported by LBDR, but 7 is not.

2.2.4 Cycle-Free Contour Fault-Tolerant Routing Algorithm

In Zhang, Greiner, and Taktak [15], the authors present a fault-tolerant, distributed, reconfigurable routing algorithm that can be used in any 2D-Mesh NoC.

The proposed routing algorithm is low cost (only an 8% hardware overhead on the NoC), it tolerates any single faulty router topology, it is scalable because its hardware cost does not depend on the mesh size, it is deadlock-free by adopting a cycle free approach, and it is deterministic to guarantee the in-order delivery.

The main idea of this routing algorithm is to route packets through a cycle-free contour surrounding a faulty router in order to bypass it. This mechanism divides the mesh in two disjoint regions as shown in Figure 2.2: the region Ⓑ containing the faulty router and its contour, and the region Ⓐ containing all the other fault-free routers. The contour of a faulty router contains all its direct neighbors (N, S, W, E), and its indirect neighbors (NW, NE, SW, SE).

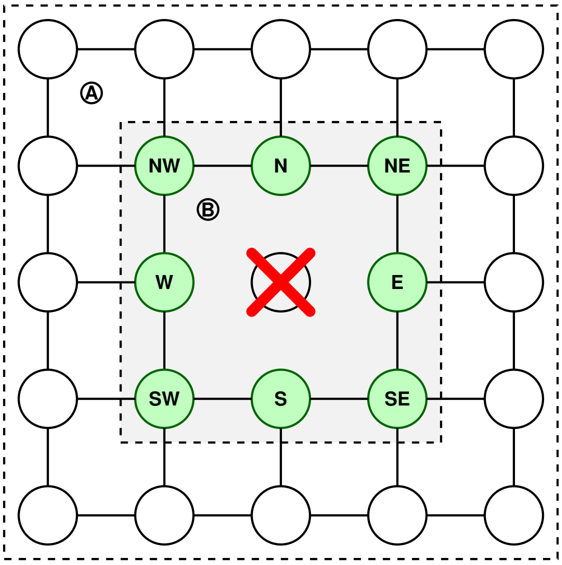


Figure 2.2 – Contour of a Faulty Router

The routers in region Ⓐ use the X-first routing algorithm, but the routers in region Ⓑ use a modified routing algorithm. This modified routing algorithm is based on the X-first algorithm but authorizes or prohibits some turns to allow bypassing a faulty router without introducing any cycle. This last property is important to guarantee a deadlock-free routing. There are nine different kinds of contours depending on the position of the faulty router, as shown in Figure 2.3. To bypass a faulty router, the modified routing algorithm allows some additional turns (with respect to the X-first algorithm) in the contour of this faulty router. Additionally, when the faulty router is in the center of the mesh, some turns are prohibited in the NE router to prevent the introduction of a cycle.

In the X-first routing algorithm, each router forwards the packet based on its local coordinates, and the destination coordinates. However, in order to implement the proposed reconfigurable fault-tolerant routing algorithm, the routers also needs a

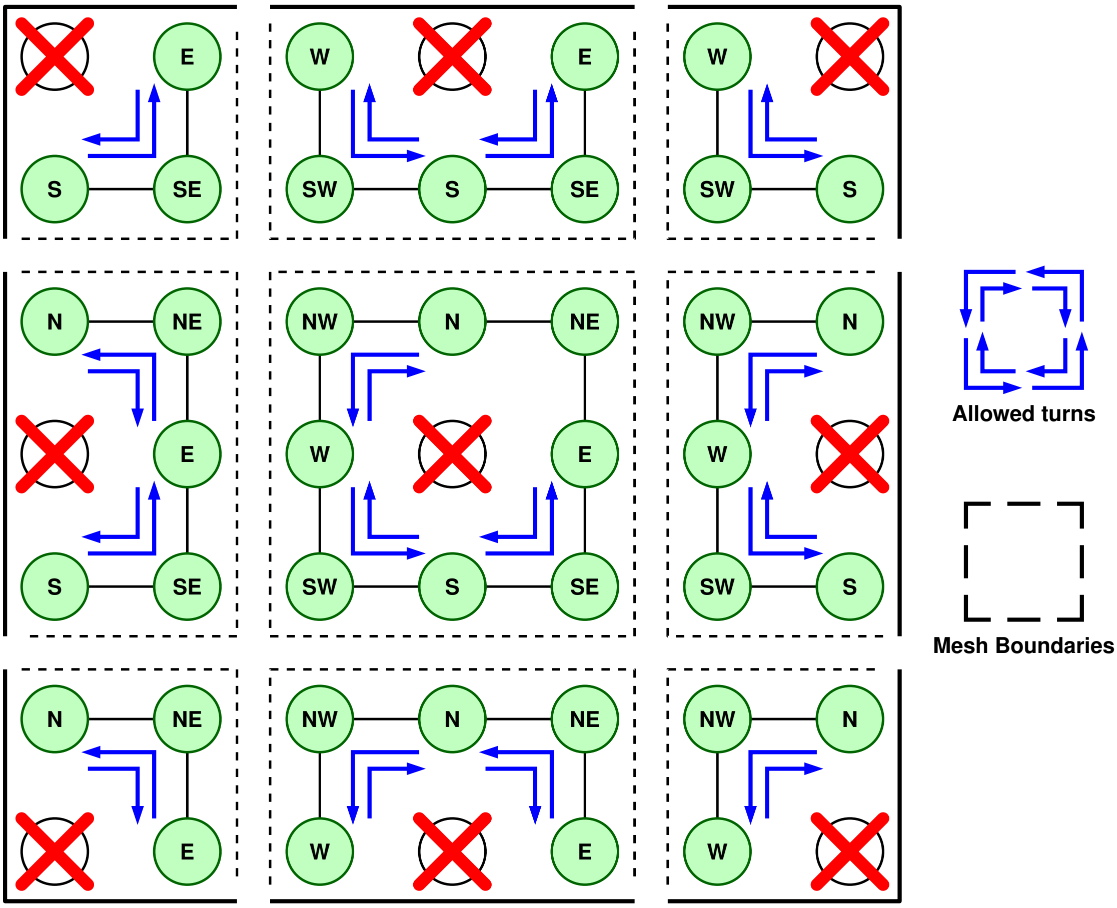


Figure 2.3 – The Nine Contour Types for Single-Faulty-Router Topologies

configuration register. This register uses 4 bits, and contains one of the following nine values: NORMAL, N_OF_X, NE_OF_X, E_OF_X, SE_OF_X, S_OF_X, SW_OF_X, W_OF_X, NW_OF_X. When the configuration register of a router contains NORMAL, it is in region Ⓐ and uses the normal X-first algorithm. Otherwise, the router is in region Ⓑ and uses the modified routing algorithm, which depends on its position with respect to the faulty router. If the value is N_OF_X, the router is at the north of the faulty router; if the value is NE_OF_X, the router is at the northeast of the faulty router; and so on. The reconfigurable routing function implemented at each router is detailed in Appendix A.

This routing algorithm supports any single faulty router topology, and can be extended to support a single faulty region: a rectangular region covering all faulty routers. However, in this region there may be functional routers which would be wasted.

This is an interesting low-cost solution to support permanent failures in the NoC. However, this solution only defines the fault-tolerant routing algorithm, and requires that the faulty router is precisely located, and it requires a reconfiguration communication infrastructure to reliably configure the routers on its contour, and repair the NoC. These two problems must be solved.

2.3 Conclusion

In this chapter, we analyzed the state-of-the-art research regarding fault-tolerance mechanisms dealing with permanent failures in many-core architectures.

These state-of-the-art solutions are either focused on supporting faulty cores, or faulty routers or communication channels in the NoC. However, these solutions do not handle the more general problem of discovering and deactivating all kinds of faulty components (cores, distributed memory banks and NoC routers), centralize this information, reconfigure the NoC, and provide the OS with an explicit cartography of an operational hardware platform. This is essential in order to allow commodity OSs (like Linux or NetBSD) to self-adapt, and run in a degraded mode despite the presence of hardware faults.

In this work, we are interested in a complete solution allowing shared-memory many-core processors to self-reconfigure, define a degraded mode, and boot a commodity OS, in the presence of hardware faults, without any external intervention. For this purpose, we propose a fully distributed, yet cooperative software-based solution, which is executed at the chip power-on by internal cores, to discover, deactivate and reconfigure the processor in the field.

Regarding the NoC reconfiguration, we choose to use the fault-tolerant routing algorithm proposed by Zhang, Greiner, and Taktak [15] in our complete solution to support permanent failures in many-core processors. However, as described above, this solution needs the faulty router to be located, and a reliable communication infrastructure to reconfigure the NoC itself. Our proposed software-based solution will intend to solve both problems.

Chapter 3

Distributed Recovery Firmware

Contents

This section introduces the proposed in-the-field self-reconfiguration mechanism to support permanent failure recovery in cache-coherent shared-memory many-core architectures. Its objective is to improve both the manufacturing yield and the lifetime of this kind of chip by implementing dynamic reconfiguration at every processor reset. We believe that the processor is capable of operating with a limited performance degradation despite the presence of faulty devices.

In order to decrease the silicon cost, this mechanism is mostly implemented as a distributed firmware executed by the internal cores. To allow in-the-field recovery, a part of this firmware is replicated in internal, distributed, Read-Only Memories (ROMs). Another part is stored in an external mass storage device to reduce the size of the internal ROMs. To reduce the risk of fault propagation, the firmware is replicated in every cluster. This allows also to improve the robustness of the solution because a faulty ROM only affects the cluster containing it.

As a consequence of using the internal cores to execute the replicated firmware, this reconfiguration firmware must execute on unreliable hardware devices because some processor cores, memory banks, or communication resources can be faulty. Therefore, the fault-tolerance mechanism relies on distributed algorithms that are executed cooperatively by the cores, and face the problem of unreliable hardware by implementing self-diagnostic procedures to detect faulty behaviors.

3.1 Global Procedure

The fault-tolerance mechanism consists of several distributed and cooperative algorithms to locate the faulty devices, centralize the distributed information to build a global map of the functional infrastructure, and reconfigure the hardware to bypass the faulty devices. The mechanism acts as a smart distributed boot-loader service, which launches a commodity Operating System (OS) upon the functional infrastructure from an external mass storage device, and provides to this last the global map of the functional infrastructure, so it adapts to the modified topology. This functional topology can change between successive executions of the OS because new permanent faults can appear at each reset.

The complete fault-tolerance mechanism is shown in Figure 3.1. It implements the following four stages:

① Hardware-Based NoC Fault Detection.

② Distributed Software-Based Fault Location.

③ Hardware-Assisted NoC Reconfiguration.

④ OS loading.

The first stage performs a fully-distributed hardware Built-In Self-Test (BIST) method for each router of the Networks-on-Chip (NoCs). The stages ②, ③, and ④ are the ones implemented in the recovery firmware.

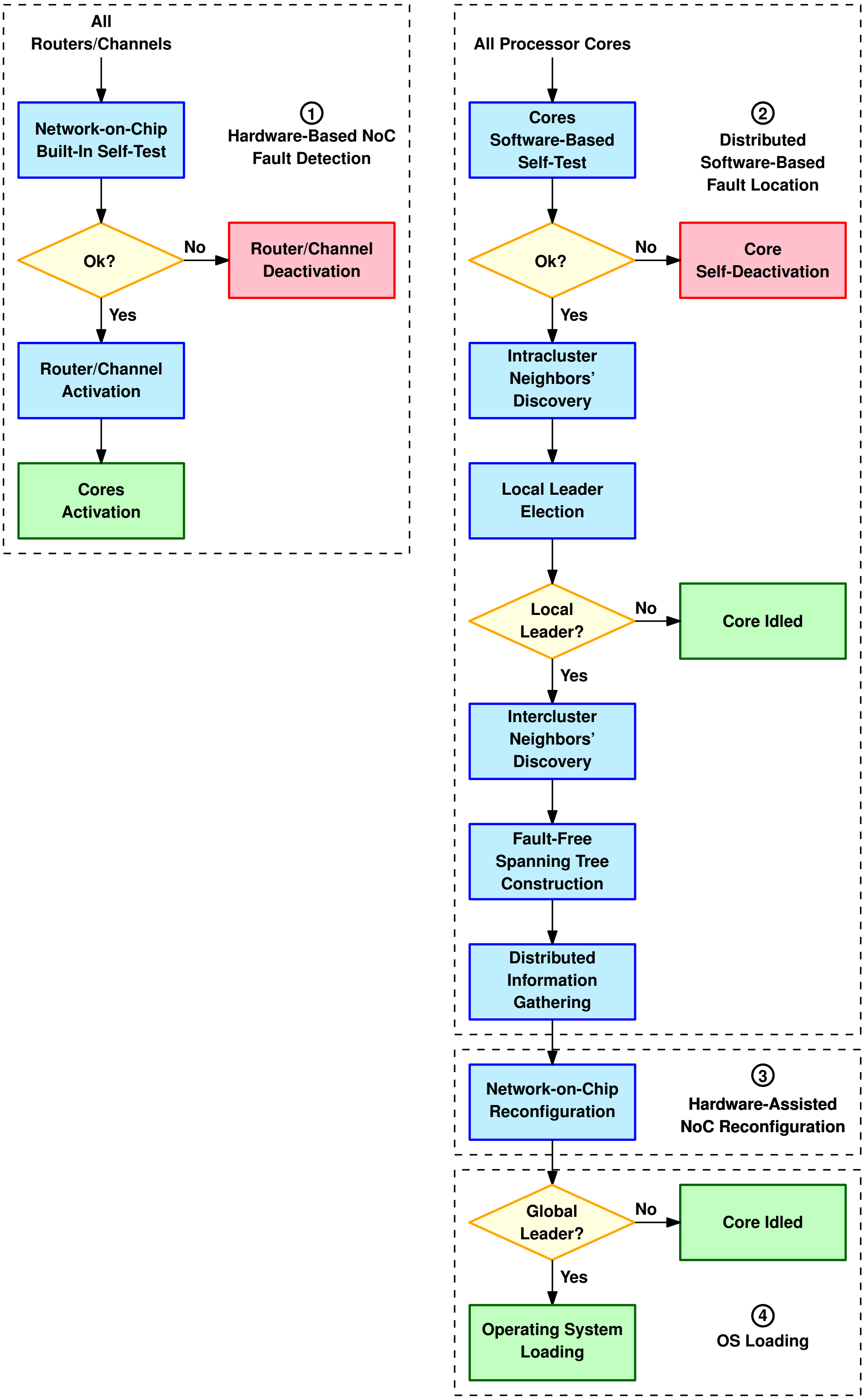


Figure 3.1 – Global Procedure Flow Diagram

3.2 Hardware-Based NoC Fault Detection

This first stage uses an existent hardware BIST solution proposed by Zhang, Greiner, and Benabdenbi [11]. This solution consists in a BIST that is distributed in every router of the NoC. The NoC is described as a set of routers connected by point to point communication channels. The BIST performs the test of each inter-router channel, and tests the router itself at each hardware reset. The BIST deactivates the faulty channels, but when the router itself is faulty, all channels connected to this faulty router are deactivated.

A deactivated channel behaves as a *black-hole: all incoming packets are consumed, and no packet is produced.*

The TSAR architecture has five NoCs in the computational layer, and each cluster has thus five routers (as detailed in Section 1.4.2). Each of these NoCs implements this BIST mechanism. When one or more of these routers in a cluster are faulty, the entire cluster is deactivated (i.e. local cores are not activated). Even if a cluster is deactivated, its functional NoC routers can still be used.

This BIST supports GALS NoCs, and its latency depends on the ratio between two clock frequencies in two neighbor routers. When the clock frequencies are the same (even with different phases), the latency is about 300 clock cycles.

3.3 Distributed Software-Based Fault Location

This stage performs the construction of a reliable software-based communication infrastructure with a tree topology, called Fault-Free Spanning Tree (FFST), which is temporarily used during the boot. This infrastructure (detailed in Chapter 4) is used to build a global map of the operational hardware components, and to reconfigure the NoCs.

Each node of the FFST is a functional cluster of the processor, and each edge is a software-based full-duplex point-to-point communication channel between two neighbor clusters. Each cluster in the FFST is represented by one of its functional local cores (called local leader), and the root of the FFST (called global leader) is a core in an Input/Output (IO) cluster.

The construction of the FFST is performed by the execution of the distributed recovery firmware at each cluster. This construction needs various phases:

1. In order to belong to the FFST, a cluster must be functional, so it needs to pass various software-based diagnostic tests (called intracluster tests) executed by the local processor cores. Then, each core discovers its functional intracluster neighbors (i.e. other functional local cores), and one of these cores is elected local leader. This intracluster stage is detailed in Section 4.1.

2. The FFST is based on the communication between neighbor clusters, therefore each local leader needs to discover with which direct neighbors a full-duplex communication is possible. A full-duplex communication is possible when two neighbor clusters can communicate through all the five NoCs in the computational layer. This intercluster neighbor discovering is detailed in Section 4.2. Section 4.3 details a specific hardware mechanism to test the coherence NoCs.

3. At this point, the local leaders know the local functional cores, and the functional neighbor clusters, but the global map of the operational resources is unknown. In order to build this global map, all local leaders build the FFST by cooperatively executing a distributed algorithm. At the end of this procedure, the global leader of the processor is elected. The construction of the FFST is detailed in Section 4.4.

4. At the end of the FFST construction, the global leader has a global map of the computational resources (cores and memory banks), but a global map of the communication resources is still not available. Therefore, the global leader uses the FFST to ask the local leaders to execute a software procedure to locate the holes in the NoCs, and then centralize the distributed information to build this global map of communication resources. This procedure is detailed in Section 4.5.

Chapter 4 details the different distributed procedures in this stage, enumerates the properties that these procedures must satisfy, and proves these properties.

Figure 3.2 shows an example of a FFST for a 4 × 3 mesh. In this example, the router in cluster **(2,1)** is faulty, and this cluster is not reachable by its neighbors. Therefore, the FFST covers all clusters but that one. As we can see, an edge of the FFST is implemented by two software mailboxes (memory buffers) to provide the full-duplex communication between two neighbor clusters.

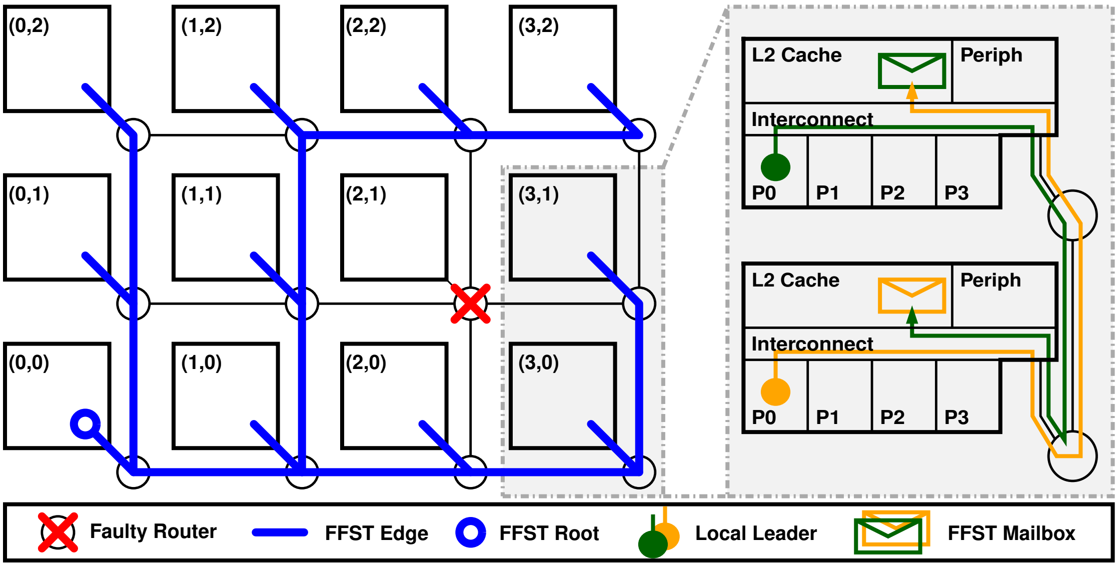


Figure 3.2 – FFST Example in a Mesh with a Faulty Router

3.4 Hardware-Assisted NoC Reconfiguration

This stage (detailed in Chapter 5) performs the reconfiguration of partially defective NoCs in order to modify the global routing function, and bypass the holes.

In order to support the NoCs reconfiguration, the routers in the NoCs of the computational layer implement the reconfigurable routing algorithm defined by Zhang, Greiner, and Taktak [15] (detailed in Chapter 2). All routers contain a reconfiguration register that needs to be written to change the routing function, and we propose a software procedure to perform this reconfiguration (detailed in Section 5.2).

During the NoC reconfiguration phase, the global leader uses the global map of the communication resources to determine the location of holes, and it uses the FFST as a reconfiguration bus to reach reliably the clusters on the contour of holes, and reconfigure the involved routers. During this procedure, the local leaders behave as software-based routers allowing the global leader to address any functional cluster.

At the end of this stage, the partially defective NoCs are reconfigured, and the hardware communication between functional clusters is reestablished. Therefore, the FFST is not needed anymore for intercluster communications. Figure 3.3 shows an example where a core in the cluster **(0,2)** accesses the memory in the cluster **(2,0)**. Due to the reconfiguration of the routers on the contour (in gray), the command bypasses the faulty router **(2,1)**.

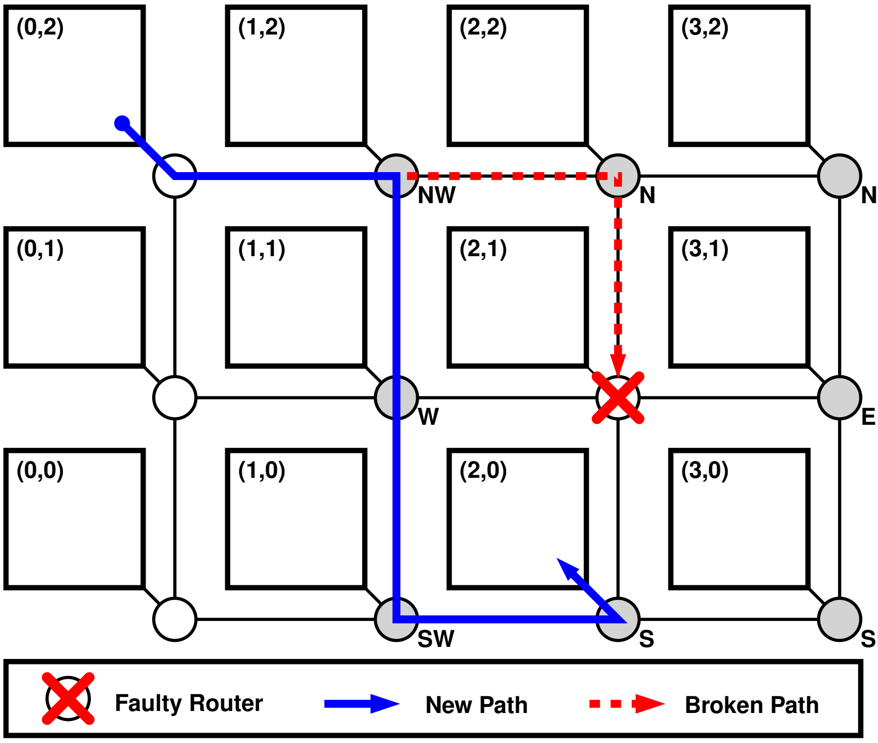


Figure 3.3 – NoC Reconfiguration Example with a Faulty Router in Coordinates **(2,1)**

3.4.1 Broadcast Support With Holes

The reconfigurable routing algorithm used supports any single-faulty-router topology for unicast communications. However, the M2P NoC of the TSAR architecture, in addition to unicast communications, uses broadcast communications to allow L2 cache controllers to invalidate a cache line in all L1 cache controllers.

As broadcast communications are not supported by this reconfigurable routing algorithm, we propose a new routing algorithm supporting broadcast on any single-faulty-router topology (detailed in Section 5.4.1).

3.4.2 3D NoCs Reconfiguration

Additionally, the TSAR architecture has other NoCs above the computational layer with a 3D-mesh topology, and we propose an extension to the reconfigurable routing algorithm to support this kind of topology in Section 7.5.

We propose also a software-based procedure to locate the holes in these NoCs, and perform their reconfiguration. Chapter 7 explains these procedures to deal with failures in 3D NoCs.

3.4.3 Memory Segment Reallocation

The TSAR architecture implements a physically distributed address space, where each cluster controls one physical memory segment, and when a cluster is deactivated, the corresponding physical memory becomes unusable.

As TSAR has redundant communication resources provided by L2-L3 interconnects, we propose a mechanism to modify the global routing function to reallocate the physical memory segment of a deactivated cluster to one of its neighbors (detailed in Section 5.3), to avoid the memory loss.

3.5 OS Loading

During this stage, the global leader loads in memory the OS from an external mass storage device.

Commodity OSs, like Linux or NetBSD, can use a structure called Device Tree Blob (DTB) to describe the hardware platform ([26], [27]). This structure is passed from the boot-loader to the kernel. The DTB provides the kernel with the ability to adapt to different configurations of the same architecture with no need of recompilation.

This work proposes to store in the external mass storage a reference DTB containing all the devices in the processor. This reference DTB is copied on memory, and patched by the boot-loader in order to remove the faulty devices from it (e.g. processor cores, memory banks). The devices to remove during the patch are determined from the global map of operational computational resources built during the distributed software-based fault-location phase.

Finally, the patched DTB is passed to the OS as a parameter, and the OS has the ability to adapt to the possibly degraded architecture at every boot.

3.6 Conclusion

In this chapter, we have presented a global description of our in-the-field fault-tolerance mechanism to support permanent failures in shared-memory many-core architectures. It consists in a distributed recovery firmware (stored in distributed on-chip ROMs), which acts as a smart distributed boot-loader, and launches a commodity operating system upon the functional hardware infrastructure.

This fault-tolerance mechanism uses pre-existent technologies: the NoC BIST [11], and the NoC reconfigurable routing algorithm [15].

Our main contribution is to include these technologies in a complete flow to support in-the-field permanent failures. The internal cores execute, in an unreliable hardware, self-diagnostic software procedures to deactivate the faulty hardware devices, and reconfigure the architecture. These procedures implement distributed algorithms to:

1. Elect a local leader at each cluster.
2. Elect the global leader of the processor.
3. Build the reliable FFST software-based communication infrastructure.
4. Build the global map of the operational hardware devices.

Chapter 4

Distributed Fault-Location

Contents

This chapter presents the software-based method to locate the faulty cores, faulty memory banks and faulty routers, and build a global map of the operational platform. This global map describes the actual Network-on-Chip (NoC) topology (as it can exist black-holes in the 2D mesh), and the different hardware resources that are reported functional to the Operating System (OS). As described in Chapter 3, the method makes the assumption that all faulty routers have been deactivated by the NoC Built-In Self-Test (BIST).

To build this global map, the proposed solution introduces the Fault-Free Spanning Tree (FFST) infrastructure. This infrastructure provides a temporary software-based reliable communication support between the functional clusters of the architecture. It is built dynamically at each hardware reset of the processor by the execution of a distributed recovery firmware at each cluster. The construction of the FFST is performed in the following three phases, which are explained in the following sections:

1. Intracluster phase (Section 4.1).
2. Intercluster phase (Section 4.2).
3. Fault-Free Spanning Tree's construction (Section 4.4).

The method has been implemented in the many-core TSAR. However, it supports any shared-memory many-core architecture implementing a 2D mesh topology. And it can be extended to support any multidimensional mesh or torus topology.

4.1 Intracluster Phase

At the end of the NoC BIST, the hardware reset signal of the cores is deasserted. In each cluster, all local cores start to execute the recovery firmware stored in the local Read-Only Memory (ROM). The cores test the local hardware resources to decide if the cluster should be declared functional. This test consists in several diagnostic software procedures. A functional cluster needs to satisfy the following conditions:

1. At least one local core is functional.
2. The local ROM containing the recovery firmware is functional.
3. The local memory bank is functional.
4. All the local interconnects are functional.

When the processor cores start the execution of the recovery firmware, the cluster is logically disconnected from the global NoC: only local communications are possible (local-to-global or global-to-local communications are blocked). This disconnection is achieved through the introduction of a hardware barrier mechanism in the gateway between the local interconnects and the NoC routers (detailed in Section 4.1.4). The goal of this mechanism is to avoid the fault propagation (e.g. when a faulty core triggers invalid transactions because of a faulty control signal).

Additionally, the L2 cache controller implements a special mode (called scratchpad mode) at hardware reset. In this mode, the L2 cache controller behaves as a simple local Random Access Memory (RAM), because read and write transactions to the external RAM are inhibited. As for the hardware gateway barrier, the goal of this mechanism is to avoid the fault propagation: avoid that a faulty core writes the external memory. In TSAR, the local L2 cache controller has a capacity of 256 KBytes. Therefore, the distributed recovery firmware can only use the lowest 256 KBytes of the memory space at each cluster. If a faulty core reads or writes beyond this limit, the L2 cache drops the request, and do not respond to the core. This emulates a black-hole behavior, and the faulty core is blocked because the transaction never completes.

The hardware gateway barrier and the special scratchpad mode of the L2 cache controller are both controllable by the software. The gateway barrier is disabled only if the cluster is declared functional at the end of this intracluster phase. And the special scratchpad mode of the L2 cache controller is disabled after the NoC reconfiguration phase.

At the end of the intracluster phase, if a cluster does not respect one of the above conditions, it is declared faulty and kept disconnected from the global NoC. Processor cores in a faulty cluster enter an idle mode and do not participate in further stages of the recovery process. All faulty components (cores included) will not be reported to the OS and will not be used during the OS's execution. Moreover, because of the hardware gateway barrier, the entire cluster behaves as a black hole. Even if a cluster is disconnected, its NoC routers can still be used if they are functional.

The intracluster phase consists in the following sequentially executed procedures:

1. Software-Based Self-Test (SBST).
2. Intracluster Neighbor Cores' Discovery.
3. Local Leader Election.

4.1.1 Software-Based Self-Test (SBST)

The SBST contains procedures which test the processor core's internal logic and registers, and the L1 caches. If a core self-diagnoses as faulty, it self-deactivates. A self-deactivated core is logically disconnected from the architecture: it enters an idle mode, and it is not reported to the OS.

During this SBST, the cores use the local memory to store temporary data, and the local interconnects to communicate with the local memory. Additionally, they read instructions and constant data from the local ROM. As these components cannot be trusted, they are tested, at the same time as the cores, by the cores.

Property 4.1.1. *At the end of the SBST stage, each processor core that is faulty or detects a fault in a cluster component (local memory, local interconnects or local ROM) self-deactivates.*

Test of the Local Memory and Direct Interconnects

When processor cores start the execution of the recovery firmware, each one uses a private exclusive memory segment for its stack. In order to test the local memory, each core writes a sequence of patterns $P = p_0, p_1, \ldots, p_n$ in its stack segment, and then checks that each of these patterns can be correctly read.

This write, read & verify approach to test memories is called *March* tests [28]. A *March* test is a sequence of memory operations (read & write) applied to each cell of the target memory. For example, if only stuck-at faults are considered, the cores use two patterns: a pattern p_0 and its complement $\overline{p_0}$. Such procedure allows verifying that each bit of a memory location is not stuck at some value (an example is shown in Figure 4.1). In this process, the data bus of direct local interconnects (used by normal memory accesses) is also tested. If a fault is detected by a processor core during this test, the core self-deactivates.

In TSAR, during a data write or read, the data bus of the *CMD* and *RSP* local interconnects are respectively tested.

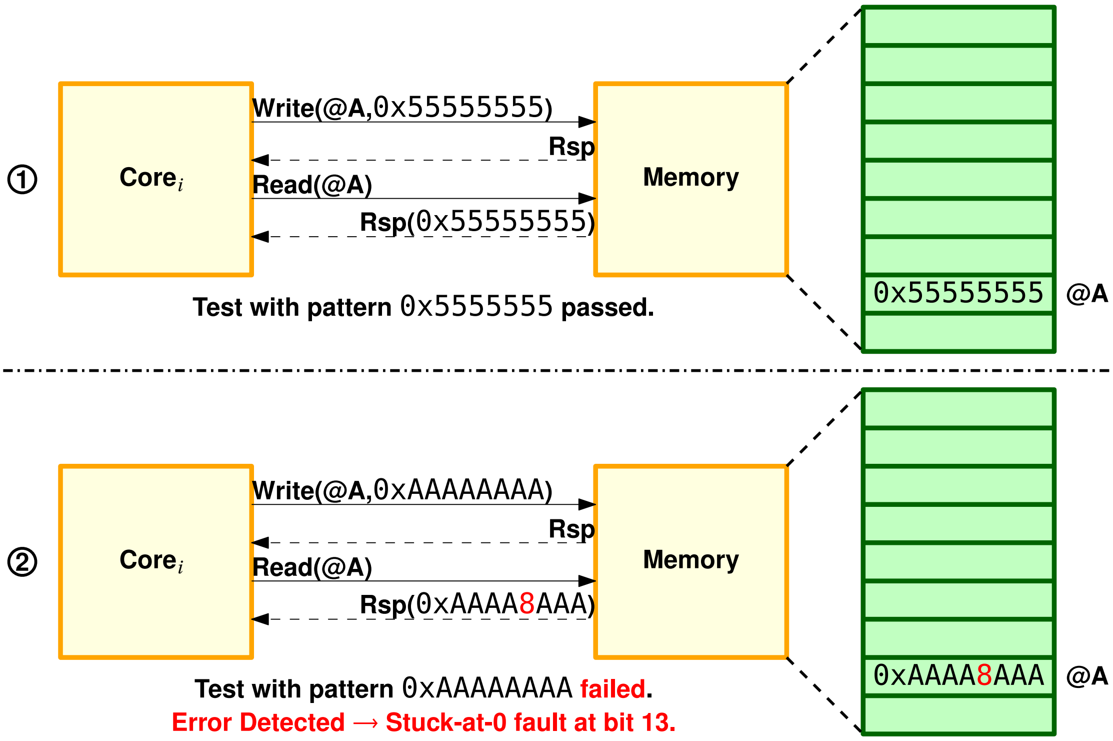


Figure 4.1 – Software-Based Memory Test: Processor Core Writing Two Test Patterns to Detect Stuck-at Faults

At the end of this test, each core has tested both the memory and the direct local interconnects. The memory is only partially tested because each core only tests its stack segment to avoid conflicting accesses with other cores. Therefore, in next stages the local memory should be tested more extensively.

Test of the Coherence Local Interconnects (*P2M, M2P, CLACK*)

In TSAR, three specific NoCs are used to maintain the coherence between L1 and L2 caches.

During the intracluster phase, the processor cores in all clusters test these local coherence interconnects, using a specific hardware mechanism, which is detailed in Section 4.3.

Test of the Local ROM

The local cores read a subset of words in the local ROM and perform a hash operation to produce a signature. Then, a reference signature stored in the ROM is compared with the resulting signature. If the signatures are different, the cores self-deactivate.

4.1.2 Intracluster Local Neighbors' Discovery

This discovery procedure is only executed by the functional cores because, as stated by Property 4.1.1, each faulty core self-deactivates during the SBST stage.

In this procedure, each core executes the Algorithm 4.1, which is detailed below, in order to discover the other functional cores in the cluster. This procedure must satisfy the following property (proved in Section 4.1.2):

Property 4.1.2. *At the end of the procedure, each functional core i has a list NP_i with all the functional cores in the same cluster (local neighbors).*

The local neighbors' discovery procedure is executed in parallel by all processor cores. Additionally, the cores may start asynchronously the execution of this procedure, because some cores may finish sooner the SBST stage.

Algorithm 4.1 receives as input a list NP_i containing all possible neighbors in the same cluster of the executing core i. The first step is to set a status variable to *OK*. Each core i has a status variable $status_i$ that can only be written by core i, but can be read by the other cores. Then, each core checks the status variable of its possible neighbors. There are two reasons why a status variable would not contain the *OK* value: (1) the associated processor core is faulty, and it has self-deactivated during the SBST stage, or (2) the associated processor core is functional but it has not finished yet the SBST. Therefore, a processor core checks at most *MAX_RETRIES* times each neighbor's status variable. If this maximum number of retries is reached for a neighbor, this neighbor is removed from the NP_i list. Otherwise, if a neighbor's status is *OK*, the core keeps this neighbor in its own NP_i list, and adds it to the *DONE* list so this neighbor is not tested anymore (the neighbor has been discovered).

When a core finishes the execution of this procedure, the list *NP* contains the *IDs* of all functional cores in the same cluster.

Algorithm 4.1: Local Neighbors' Discovery

```
LocalNeighborsDiscovery()
Input: A list $NP_i$, built by core $i$, containing all neighbor cores in the same cluster
Result: $NP_i$ contains the functional local neighbor cores
begin
    /* Inform neighbors that core $i$ is functional */
    Write($status_i$, $OK$)

    /* The $DONE$ set will contain the functional neighbors */
    $DONE \leftarrow \emptyset$

    /* Set to $0$ the number of retries for each neighbor */
    foreach $n \in NP_i$ do $retries_n = 0$

    while $DONE \neq NP_i$ do
        foreach $n \in \{NP_i - DONE\}$ do
            if Read($status_n$) = $OK$ then
                $DONE \leftarrow DONE + \{n\}$
            else if $retries_n \geq MAX\_RETRIES$ then
                $NP_i \leftarrow NP_i - \{n\}$
            else
                $retries_n = retries_n + 1$
            end
        end
    end

    /* Complete the $NP_i$ list with the executing core $i$ */
    $NP_i \leftarrow NP_i + \{i\}$
end
```

Initialization value of the status variables

In order to avoid false diagnostics (consider a faulty core as functional), the status variables of all processor cores must be initialized to a value different from the *OK* value before the execution of the neighbors' discovery procedure. This guarantees that if a core reads an *OK* value on a status variable, it is because the associated processor core has written it.

The initialization of these variables is not a simple problem because all cores execute concurrently the local firmware. For this reason, the memory locations used for the status variables are initialized by the hardware reset.

In TSAR, there are four processor cores per cluster, and the procedure needs four hardware initialized locations in each cluster. The Extended Interrupt Controller Unit (XICU) peripheral provides up to 32 memory-mapped registers for inter-core communication. These registers, initialized to 0 by the hardware reset, are used for the status variables, and we choose another value for *OK* (e.g. 0xFFFFFFFF).

Proof of the Local Neighbors' Discovery Procedure's Property

This section proves the Property 4.1.2 of the neighbors' discovery procedure. This proof uses two auxiliary lemmas explained hereafter:

Lemma 4.1.2.1 (Liveness)**.** *Each functional core remains in the NP list of all its functional neighbors.*

Proof. Each functional core *i* eventually writes in its $status_i$ variable the *OK* value. Then, its functional neighbors eventually read the $status_i$ variable, and they keep the core *i* in their *NP* list. Of course, the *MAX_RETRIES* value must be chosen big enough to allow all functional cores write their status variable. ■

Lemma 4.1.2.2 (Safety)**.** *At the end of the procedure, the list NP of each core contains only functional cores.*

Proof. When a core *i* is faulty, it self-deactivates during the SBST (Property 4.1.1), and it does not set its status variable ($status_i$) to *OK*. Therefore, the other processor cores never read the *OK* value on the $status_i$ variable, and they remove the core *i* from their list *NP*. ■

Lemma 4.1.2.1 and Lemma 4.1.2.2 prove that functional cores remain in the *NP* list and faulty cores are removed. This proves the Property 4.1.2.

Definition of the MAX_RETRIES Constant

An important characteristic of this algorithm is that the verification of the property above depends strongly on the *MAX_RETRIES* constant. This constant can be bounded because the local interconnect implements a round-robin policy.

During the intracluster phase, the processor cores share the same local ROM, memory and peripherals. Figure 4.2 shows partially the *CMD* local interconnect of the TSAR architecture (only the processor cores, L2 cache and the XICU components are showed). The arbiters in the local interconnect implement a round-robin policy guarantying that all cores access the resources in a bounded time. Therefore, if all cores start the execution of the local neighbors' discovery at the same time, *MAX_RETRIES* can contain a small value like 1. But, as the SBST execution time may variate for different cores we decide to choose a bigger value (16).

Another important characteristic is that the chosen value affects the overall latency of the distributed recovery firmware. The neighbors' discovery procedure finishes when all the possible neighbors have been discovered, or when the number of retries for each non-discovered neighbor reaches the *MAX_RETRIES* constant. Then, if the *MAX_RETRIES* value is big, and there is one faulty processor core, the functional processor cores in the cluster wait an unnecessarily long time.

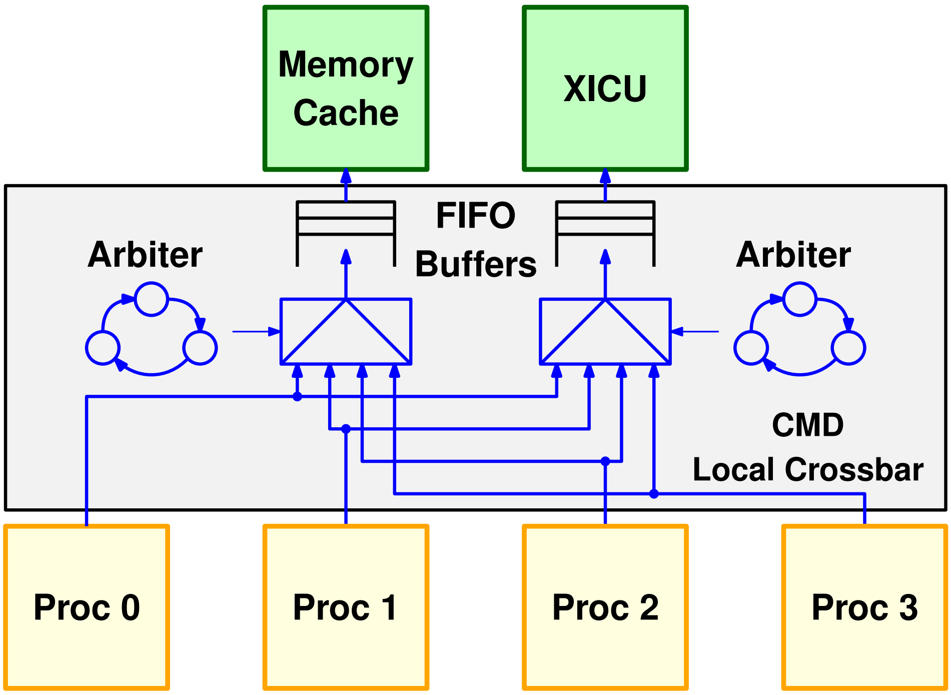


Figure 4.2 – TSAR CMD Local Interconnect: Multiplexing at Targets

4.1.3 Local Leader Election

The local leader election stage follows the neighbors' discovery procedure. During this stage, a procedure is executed at each cluster to elect a local processor core as leader of its cluster. This leader represents the cluster in further stages of the recovery firmware. The implemented procedure must guarantee the following property:

Property 4.1.3. *At the end of the local leader election, there must only be one processor core elected as local leader at each cluster.*

The proposed procedure uses a simple criterion to elect the local leader: the functional processor core with the lowest *ID* becomes the leader. Algorithm 4.2 shows the procedure executed by all the functional cores at each cluster.

Algorithm 4.2: Local Leader Election

```
LocalLeaderElection()
Input: The list $NP_i$ of core $i$ with the functional cores in the same cluster
Input: The constant $ID_i$ with the $ID$ of core $i$
begin
    /* Check if a local neighbor core has a lower ID                          */
    foreach $n \in NP_i - \{i\}$ do
        if $ID_n < ID_i$ then Idle()
    end
    /* Only one core (the local leader) exits the foreach without going to idle  */
end
```

A core i checks in its list NP_i if there is a local neighbor with a lower ID than his. If there is at least one, the core i enters an idle state. Otherwise, it becomes the local leader and passes to the next stage of the recovery firmware. Every core that enters the idle state is not used anymore during the recovery firmware execution but it will be reported as functional to the OS. Therefore, after the OS is loaded, it can use these cores.

The local leader election procedure is the last stage of the intracluster phase. After this phase, the local leader participates in the intercluster phase, but before it has to make some additional tests. As mentioned previously, each local core tests only the resources that it uses. In particular, each core tests its memory segment. However, when there are faulty cores, their segments are not tested, and the local leader has to test these segments to finally diagnose its cluster as functional (if no problem is detected). If a problem is detected, then the local leader self-deactivates, and its cluster does not participate in further stages of the distributed recovery firmware.

Proof of the Local Leader Election's Property

This subsection proves the Property 4.1.3 of the local leader election procedure. This property states that the local leader election procedure must guarantee that only one core (with the lowest local ID) is elected as leader at each cluster, and the other cores enter an idle state.

Property 4.1.2 states that at the end of the local neighbors' discovery procedure each core has a list of all functional cores in the same cluster. Then, the $NP_0, NP_1, \ldots, NP_n$ lists are the same ($NP_0 = NP_1 = \cdots = NP_n$). As each core has an unique *ID*, these lists have one unique minimum, and the corresponding core becomes the local leader.

4.1.4 Gateway Hardware Barrier

After the hardware reset, each cluster is logically disconnected from the rest of the architecture in order to improve the fault-containment. At this point, only local-to-local communications are allowed (communications between local processor cores and local peripherals). The goal of this disconnection is to avoid that a faulty initiator *pollutes* the rest of the architecture by sending invalid packets. This disconnection is implemented with a hardware barrier mechanism at each interface between local and global networks (gateway between local interconnects and NoC routers). Only if a cluster passes its intracluster tests, this barrier is deactivated.

In TSAR, this hardware barrier is implemented in the five networks as shown in Figure 4.3. This barrier is controlled by an enable signal that is itself controlled by the software. To allow this control from the software, the hardware barrier's enable signal is connected to a memory-mapped register that can be written by the software. When this register contains the value 0, the enable signal is set, and the hardware barrier is activated. If the register contains a value different from 0, the enable signal is unset, and the hardware barrier is deactivated. This memory-mapped register, in the case of TSAR, is in the XICU component. This register is initialized to 0 after the hardware reset and therefore, the hardware barrier is activated.

As can be seen in Figure 4.3, the hardware barrier uses a simple combinational circuit. It consists of 4 logic gates per network: 2 logic gates for the incoming channel and 2 logic gates for the outgoing channel. These gates are connected to the control signals of the incoming and outgoing channels to recreate a black-hole behavior. A black-hole consumes every incoming packet and no outgoing packet is produced. If a faulty local initiator tries to send a packet to another cluster when the barrier is activated, this packet is dropped at the gateway.

The intracluster phase finishes by the election of the local leader. At this point, the cluster has been tested and diagnosed as functional. Therefore, the local leader is the one in charge of deactivating the gateway hardware barrier.

4.2 Intercluster Phase

As presented in Chapter 3, the FFST is a tree covering the functional clusters of the architecture. Each node of this tree is the local leader of a functional cluster, and each edge is a full duplex communication channel between two neighbor nodes. An edge is implemented as a pair of point-to-point software mailboxes (memory buffers). After the intracluster phase, each local leader knows the local functional cores, but it does not know the functional neighbor clusters' status.

Therefore, the first step in the FFST's construction is the discovery of neighbor clusters. This discovery stage is executed after the intracluster phase. The executing cores are the local leaders, and the clusters are not disconnected anymore from the global NoC, because the local leader disabled the gateway hardware barrier.

Additionally, the faulty NoC routers behave as black-holes, and the functional ones implement the X-first routing function.

4.2.1 Intercluster Neighbors' Discovery

During this stage, each local leader tries to communicate with its direct neighbor clusters. In the case of TSAR, each local leader tries to communicate with its neighbors through the five implemented networks (*CMD*, *RSP*, *M2P*, *P2M* and *CLACK*). When a neighbor passes the test, it means that a FFST edge may be implemented between these two neighbor clusters. Therefore, this discovery procedure must guarantee the following property:

Property 4.2.1. *At the end of this procedure, each local leader i has a list NC_i containing the neighbor clusters with which a full-duplex communication is possible.*

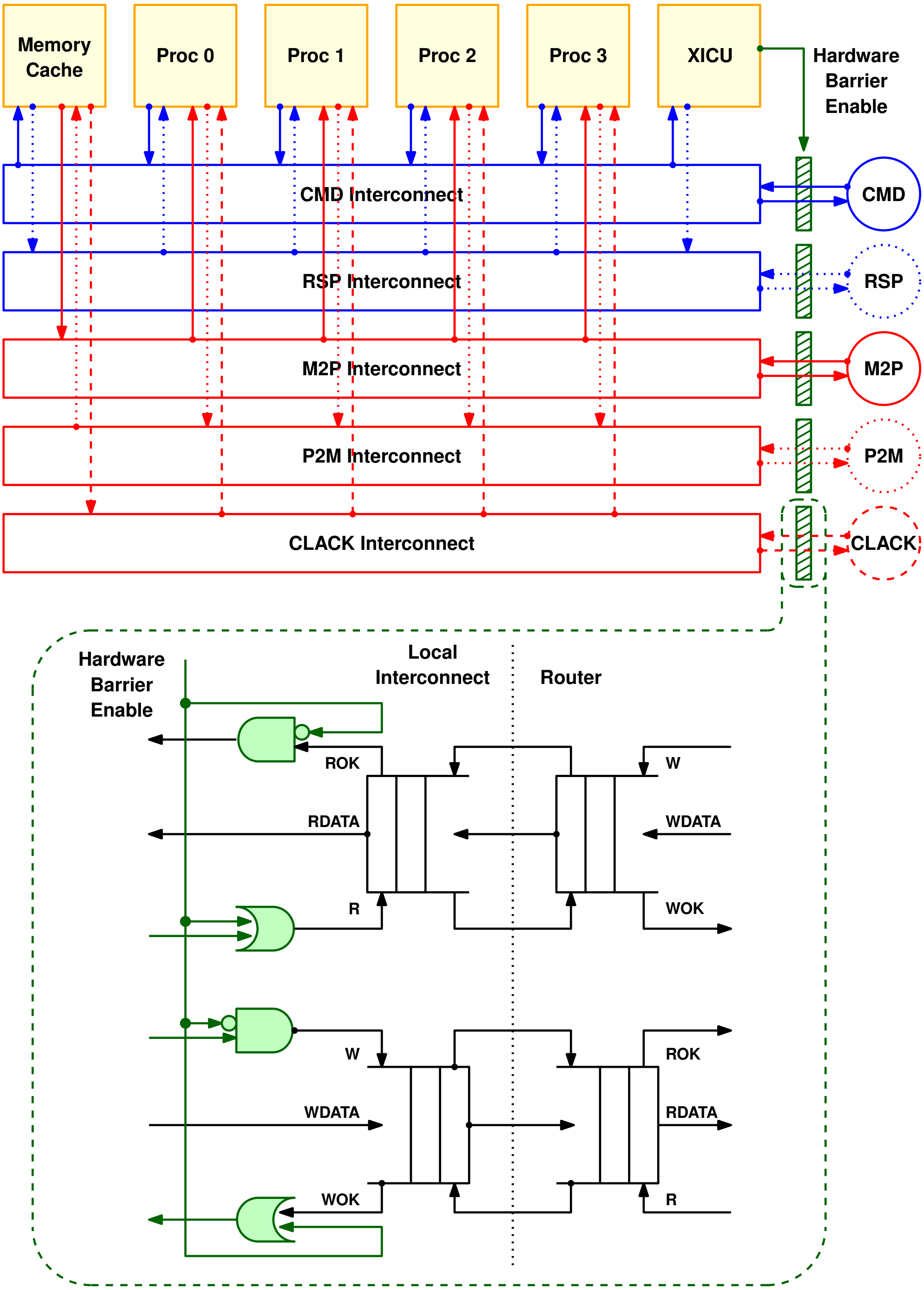


Figure 4.3 – Gateway Hardware Barrier

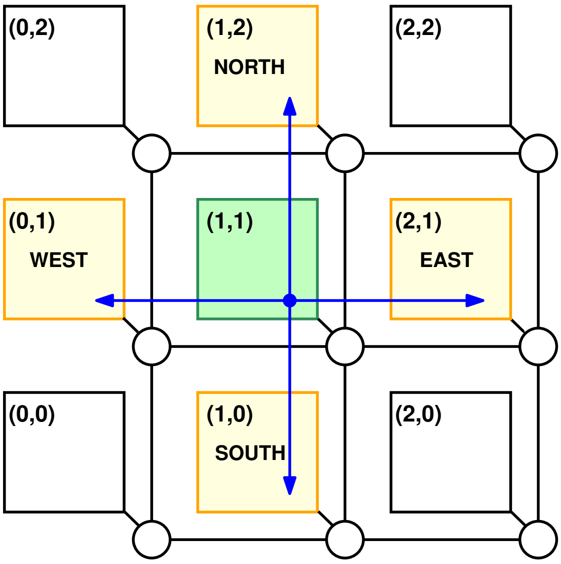


Figure 4.4 – Intercluster Neighbors

In a 2D mesh, two nodes A and B are considered neighbors when the Manhattan distance $distance(A,B)$ equals 1. This distance is computed with Equation 4.1, where A_X, B_X, A_Y, B_Y are the X and Y coordinates of nodes A and B, respectively.

$$distance(A,B) = |B_X - A_X| + |B_Y - A_Y| \tag{4.1}$$

Therefore, the number of direct neighbors of a node in a 2D mesh is four (**NORTH, SOUTH, EAST, WEST**), except for nodes in mesh's boundaries where the number of neighbors is fewer (three or two depending if it is on a side or a corner, respectively). The Figure 4.4 shows the neighbors of a central cluster in a mesh. In this example, the neighbors of cluster **(1,1)** are: **(1,2), (1,0), (2,1), (0,1)**.

Hereafter is presented the test of the *CMD* & *RSP* networks, which are used for normal memory accesses. The test of the *coherence* networks (*P2M, M2P* & *CLACK*) is discussed in Section 4.3.

In order to test the *CMD* & *RSP* networks, the neighbor clusters' discovery stage uses a modified version of the neighbors' discovery algorithm described in Section 4.1.2. Some modifications are required because there is an important difference: the executing cores are local leaders in different clusters. This difference implies that local leaders need to access not only their local memory but also the neighbor clusters' memory to test the communication.

In a shared-memory architecture, any core can access any memory location in any cluster, and the target cluster is generally identified by the Most Significant Bits (MSBs) of the address. For example, in TSAR the target cluster is identified by the 8 MSBs of the 40-bits physical addresses.

As for the intracluster phase, each local leader has its status variable in its local memory. The first action performed is to write *OK* in its local status variable. The second action is to check the status variable of each neighbor cluster by using the appropriate memory addresses. However, during this intercluster test a new problem can arise because there can be black-holes in the path.

Figure 4.5 shows the hardware components of the direct network tested during a neighbor-to-neighbor communication. When a local leader tries to communicate with a neighbor cluster, two kinds of hardware components are tested: the target neighbor cluster itself, and the NoC routers between both clusters. If one of these hardware components is faulty, it behaves as a black-hole: faulty routers were transformed in black-holes by the NoC BIST, and faulty clusters are black-holes because the gateway hardware barrier is not deactivated. The following section explains how the cores can deal with black-holes.

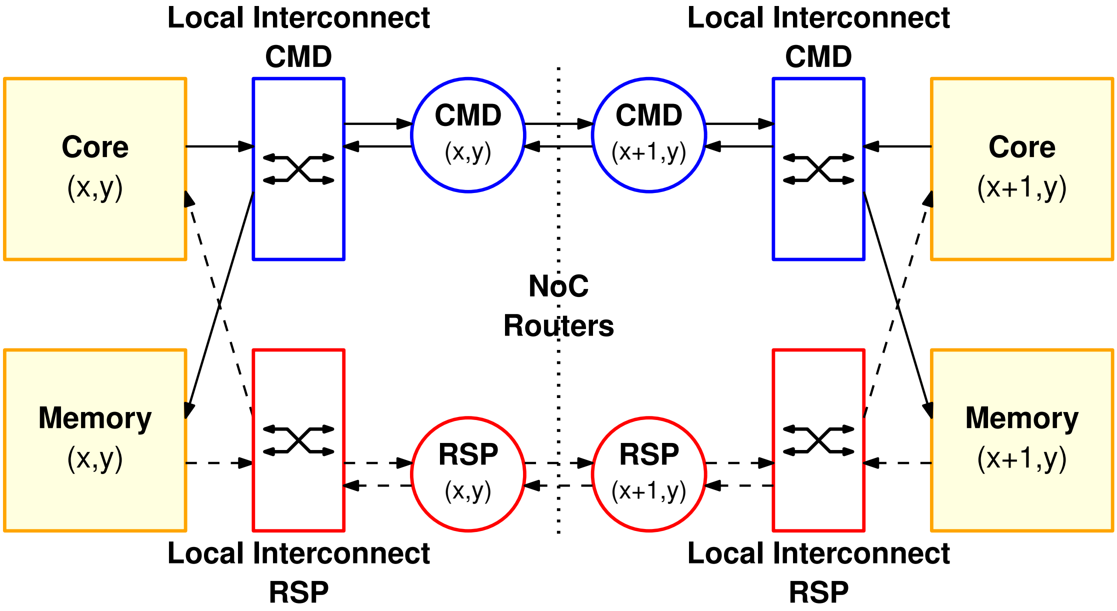


Figure 4.5 – Interconnection Path Between Neighbor Clusters

Watchdog Timer

When a core tries to read a non-cacheable memory location, the data cache controller starts a network read transaction. This transaction consists of a command packet to the device mapped on the target address, followed by a response packet from the target device to the issuing cache controller. The cache controller that started the transaction waits the response packet before allowing the core to continue the execution of further instructions. Therefore, when a local leader tries to read the status variable of a neighbor cluster, if there is a black-hole in the path, the command or response packet would be dropped, and the local leader processor core would wait indefinitely the end of the transaction.

To solve this problem, we introduce a watchdog timer in the L1 cache controller to abort the transaction. When a L1 cache controller triggers a read or write transaction, it triggers also the watchdog timer. If there is a watchdog *timeout* (the watchdog timer reaches a specified threshold), the L1 cache controller responds to the processor core with a software exception, aborts the transaction, and writes a specific value into the exception code register (**TransactionAborted**).

The watchdog threshold is stored in a dedicated cache configuration register, which is set by software. Its value must be set accordingly to the worst case transaction latency. This threshold is of course architecture dependent. In the case of TSAR, the worst case latency for a memory transaction on a neighbor cluster is less than 100 cycles. Here again, this worst case value has always an upper-bound because both the local interconnect and NoC routers implement a round-robin policy.

The software exception makes the processor core to execute the exception handler of the distributed recovery firmware. This exception handler makes a decision about the neighbor which caused the exception, and then returns to the neighbors' discovery algorithm to continue the test of other neighbors.

Neighbor Clusters' Discovery Algorithm

The modified algorithm for the neighbor clusters' discovery is shown in Algorithm 4.3. Initially, each local leader i has initially a list NC_i with all its possible neighbors (*NORTH*, *SOUTH*, *EAST*, *WEST*). During the execution, each local leader tries to read the status of the neighbors in its own list and removes the ones for which it does not succeed. At the end, the NC_i list contains only the neighbor clusters for which the local leader i read successfully the *OK* status.

The difference, between this algorithm and the one used during the intracluster phase, concerns the Read operation on the neighbors' status variable. When a local leader tries to read a neighbor's status, it sets the *abortable* variable to true in order to inform the exception handler that a watchdog timeout can arise during this operation. If there is a watchdog timeout, the exception handler is executed, and sets to false the *abortable* variable in order to inform the software that an exception has arisen. Then, the exception handler returns the execution to the instruction following the Read operation. At this point, the software tests if the *abortable* variable is false, and if it is, the core sets the return value to *KO* so the neighbor is not added to the list of tested neighbors. If there is no exception, the Read operation returns the value read from the $status_n$ variable of neighbor n, and when the return value is *OK*, the testing core i adds n to the list of discovered neighbors.

As shown in Algorithm 4.3, when there is a watchdog timeout during the test of a neighbor, this neighbor is not immediately diagnosed as faulty. Instead, the neighbor is tested several times before making a diagnostic. When a local leader i tests a neighbor cluster n, a watchdog timeout can arise for three different reasons:

1. There is a faulty command or response router between clusters i and n.
2. The neighbor cluster is faulty, and its gateway hardware barrier is activated.
3. The neighbor cluster is functional but it has not finished executing its intracluster phase, and its gateway hardware barrier is still activated.

Algorithm 4.3: Neighbor Clusters' Discovery

```
NeighborsDiscovery()
Input: A list $NC_i$ with the testable neighbor clusters of core $i$
Result: $NC_i$ contains the functional neighbor clusters of core $i$
begin
    /* Inform neighbors that core $i$ is functional */
    Write($status_i$, $OK$)

    /* This set will contain the functional neighbors */
    $DONE \leftarrow \emptyset$

    /* Set to $0$ the number of test retries for each neighbor */
    foreach $n \in NC_i$ do $retries_n = 0$

    while $DONE \neq NC_i$ do
        foreach $n \in \{NC_i - DONE\}$ do
            $abortable = true$
            $value =$ Read($status_n$)

            /* If there is a watchdog timeout during Read(), the function
               ExceptionHandler() is called, and $abortable$ is reset to false */
            if $abortable = false$ then $value = KO$
            else $abortable = false$

            if $value = OK$ then
                $DONE \leftarrow DONE + \{n\}$
            else if $retries_n \geq MAX\_RETRIES$ then
                $NC_i \leftarrow NC_i - \{n\}$
            else
                $retries_n = retries_n + 1$
            end
        end
    end
end

ExceptionHandler()
begin
    if $ExcCode = TransactionAborted$ then
        if $abortable = true$ then
            $abortable = false$
            ExceptionReturn()
        else
            Idle()
        end
        ⋮
end
```

For cases 1 and 2, a full-duplex communication is not possible with the cluster n, and the local leader i must remove n from the NC_i list. In the case 3, the local leader i should keep n in the NC_i list, but i should wait long enough so n finishes its intracluster phase. This wait is implemented with the *MAX_RETRIES* constant. A local leader tests *MAX_RETRIES* times each neighbor before diagnosing it as faulty. This last case can arrive because the distributed firmware is executed asynchronously by the different clusters in the architecture. Therefore, the *MAX_RETRIES* should consider the Worst Case Execution Time (WCET) of the intracluster phase.

Proof of the Neighbor Clusters' Discovery's Property

The proof of the Property 4.2.1 is almost the same that for the Property 4.1.2. However, there is a difference. Even when two neighbor clusters are functional, they may not communicate to each other if there are NoC black-holes between them. Therefore, when a local leader in a cluster A tests a neighbor cluster B, three different scenarios are considered:

1. B is faulty.
2. A and B are functional, and there is no NoC black-hole between them.
3. A and B are functional, but there are NoC black-holes between them.

For the first case, as cluster B is faulty, the local leader in cluster A never succeeds to read B's status variable and finishes by removing B from the NC_A list. This guarantees that, after the execution of Algorithm 4.3, all faulty neighbor clusters are removed from the NC list of all local leaders.

For the second case, the proof of Property 4.2.1 is the same that for the intracluster neighbors' discovery. As both clusters A and B are functional, they will set their status variable to *OK*, and eventually cluster A will read the cluster B's status and keep it in the NC_A, and cluster B will eventually do the same. This guarantees that two functional neighbor clusters, with no NoC black-hole between, will keep each other in their respective NC list.

For the third case, as a full-duplex communication is not possible between clusters A and B, B should be removed from the NC_A list, and cluster A should be removed from the NC_B list. This can be stated as follows:

Lemma 4.2.1.1. *If a full-duplex communication is not possible between two functional neighbor clusters A and B, then $A \notin NC_B$ and $B \notin NC_A$.*

Lemma 4.2.1.1 can be proved by contradiction. Consider that at the end of the neighbor clusters discovery procedure, $A \in NC_B$ and $B \notin NC_A$. This means that, in one hand, cluster B successfully accessed the memory of cluster A, but on the other hand, cluster A failed to access the memory of cluster B. If cluster A failed to access the memory in cluster B, then there is black-hole in the NoC. Yet, when cluster A accesses the memory of cluster B, the memory access transaction (command and response) uses the same routers than a transaction from cluster B to the memory of cluster A (as shown in Figure 4.6). Therefore, there is a contradiction. If there is a black-hole between both clusters, then cluster B cannot successfully access the memory of A.

In conclusion, each local leader i has in its local NC_i list all the neighbor clusters with which a full-duplex communication is possible.

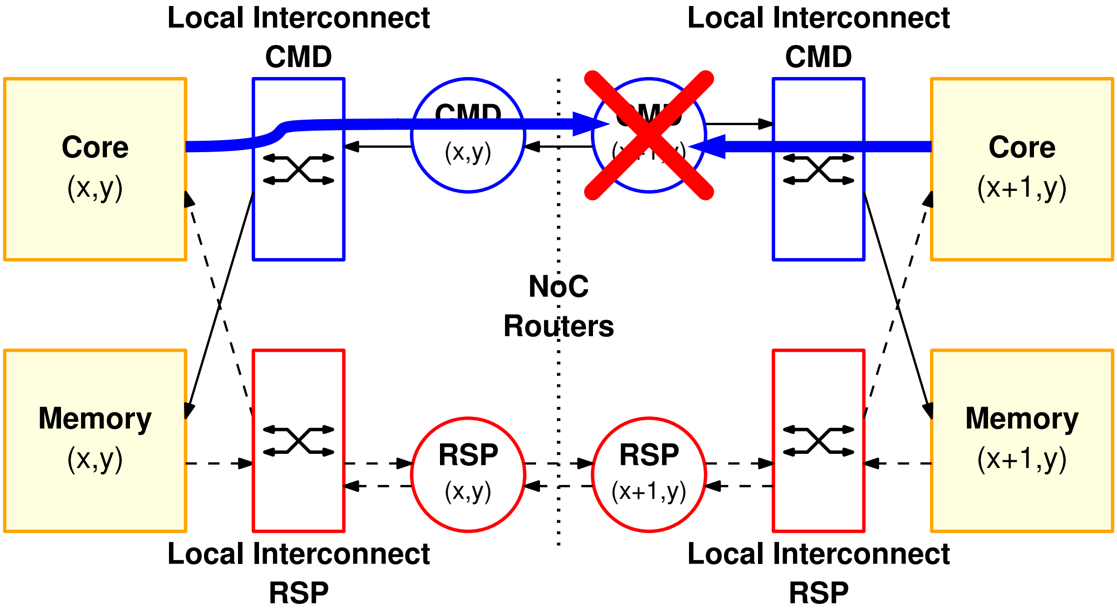


Figure 4.6 – Interconnection Path Between Neighbor Clusters (Faulty)

4.3 Coherence Networks

This section presents the proposed solution to test the coherence networks in the TSAR architecture. This solution consists in a hardware-software mechanism that allows the cores to test the coherence networks, and the test procedure is entirely driven by the software.

As described in Section 1.4, TSAR implements three NoCs for coherence transactions between L1 and L2 caches. The L1 cache controllers are initiators on the *P2M* interconnect, while the L2 caches are the targets; and the L2 caches are initiators on the *M2P* and *CLACK* interconnects, while the L1 caches are the targets. These NoCs are hierarchically implemented in two levels (as for *CMD* & *RSP* networks): local and global.

Coherence transactions are issued by cache controllers, and these transactions are not directly triggered by software reads and writes. This makes difficult the test of coherence networks and therefore, a specific hardware mechanism is proposed. To allow the software to directly test these networks, the Finite State Machines (FSMs) of L1 and L2 cache controllers must be modified. The modification allows the processor core to trigger a coherence transaction by writing into specific cache configuration registers.

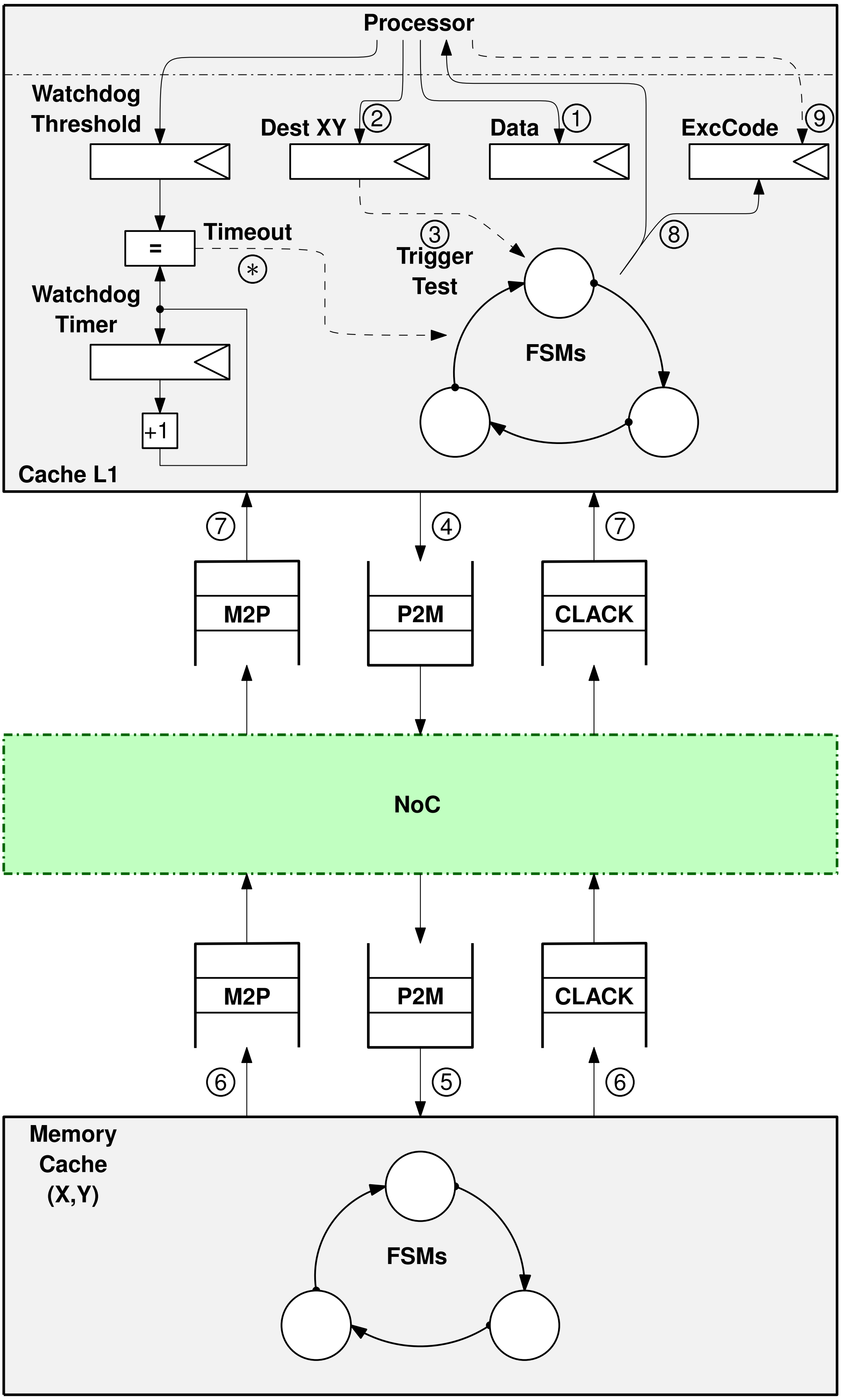


Figure 4.7 – Processor Core Triggering the Coherence Network Test

This complete coherence networks test mechanism is described in Figure 4.7. The steps are the following:

① The core writes in the **Data** register the value to be sent as data through the *P2M* network.

② The core writes in the **Dest XY** register the coordinates of the target L2 cache controller.

③ The writing in ② triggers the coherence network test and blocks the core until the test transaction is finished.

④ The L1 cache controller sends a test packet to the L2 cache controller (specified in the **Dest XY** register) through the *P2M* interconnect with the specified value (contained in the **Data** register).

⑤ When the L2 cache controller receives the *P2M* test packet, it prepares two packets by copying the received data.

⑥ The L2 cache controller sends one packet on the *M2P* interconnect and another packet on the *CLACK* interconnect.

⑦ When the L1 cache controller receives both response packets, it compares the received data with the sent one. If it does not receive both response packets before the watchdog threshold, the watchdog timer triggers a **Timeout**.

⑧ In this step, the cache controller responds to the core to release it. When the received data is different from the sent one, or if there is a watchdog **timeout**, the L1 cache controller signals an exception and writes the **TransactionAborted** value into the exception code register (**ExcCode**). Otherwise, the cache controller responds the core normally so it continues the execution.

4.3.1 Intracluster Coherence Networks Test

The test of the intracluster coherence interconnects is performed during the intracluster phase. During this phase, each processor core uses the mechanism explained above to test the coherence interconnections with the local L2 cache controller. In order to test if these networks have stuck-at faults, the cores perform twice this test with two different patterns as for the intracluster memory test. If this test fails, then the issuing core self-deactivates.

4.3.2 Intercluster Coherence Networks Test

This test is performed after the neighbor clusters' discovery procedure. Therefore, at this point, the local leaders know the neighbor clusters with which a full-duplex communication is possible through direct networks (*CMD & RSP*). These neighbors are in the *NC* list of each local leader.

As for the intracluster phase, the intercluster coherence networks test uses the mechanism explained above, but this time, local leaders try to communicate with the L2 cache of neighbor clusters (clusters in the *NC* list) instead of their local L2 cache. During this test, each local leader sends a test packet on the *P2M* network to one neighbor's L2 cache, and then waits for the acknowledgements on the *M2P* and *CLACK* networks. However, as this time the transaction passes through coherence NoC routers, if there are black-holes, one or more packets are dropped, and the local leader cache controller would wait indefinitely. As for the neighbor clusters' discovery algorithm, this problem is solved with the watchdog timer.

The software triggers the coherence test by writing in the **Dest XY** coprocessor register the coordinates of the target L2 cache, and this also triggers the watchdog timer. If both acknowledgements on networks *M2P* and *CLACK* do not arrive before the watchdog timeout, the cache controller aborts the transaction (it stops waiting for the acknowledgements), and it responds to the processor core with an error. This allows the software to diagnose the communication with the target neighbor and continue the test on other neighbors.

During the test of the coherence network, each local leader knows its functional neighbor clusters (stored in the *NC* list). Therefore, when a local leader tests the communication with one of these neighbors, if this test fails, the only possible reason is that there is a black-hole in at least one of the coherence NoCs. This means that each local leader tests the communication with each neighbor only once, and if the test fails, then it removes the target neighbor from its *NC* list because a full-duplex communication is not possible with it.

In conclusion, at the end of this coherence networks test between neighbor clusters, the *NC* list contains the neighbor clusters with which a full-duplex communication is possible through all networks (*CMD, RSP, M2P, P2M, CLACK*).

4.4 Fault-Free Spanning Tree Construction

In this phase, local leaders work cooperatively to build the FFST. The FFST is defined as a collection of nodes (clusters) and edges, where each node represents a functional cluster, and each edge is a full-duplex communication channel between two nodes. This FFST respects the following property:

Property 4.4.1. *Two functional clusters (nodes) A and B may be connected by an FFST's edge if and only if* $A \in NC_B$ *and* $B \in NC_A$.

Property 4.4.1 states that one FFST's edge can only connect two neighbor clusters which support a full-duplex communication. Therefore, the communication resources interconnecting both clusters are functional.

This software-based infrastructure is called Fault-Free Spanning Tree because it covers uniquely fault-free computation and communication resources. All nodes and edges of the FFST are mapped upon previously tested hardware components.

The root of the FFST is necessarily an Input/Output (IO) cluster, because the root cluster must access IO peripherals. When there are several IO clusters, all are candidates to become the FFST's root. The one becoming root is elected dynamically with a distributed election procedure during the FFST's construction. The local leader in the FFST's root cluster becomes the global leader, and therefore the FFST's root election problem is also called global leader's election problem.

The FFST's root is not statically defined to improve robustness: if an IO cluster is faulty, then another IO cluster can become the FFST's root. In the case where all IO clusters are faulty, the entire processor is considered faulty and therefore useless. IO peripherals support all interactions between the user and the computer. If these devices are not accessible, then the many-core processor is useless. The TSAR architecture contains two IO clusters, and therefore two candidates to become global leader.

The functional clusters belonging to the FFST will be the ones reachable from the global leader as stated in the following property:

Property 4.4.2. *Considering C_r the FFST's root cluster, and C_m a functional cluster, if C_m is* ***reachable*** *from C_r, then C_m belongs to the FFST.*

Definition 4.4.1. During the FFST's construction, a cluster C_j is **reachable** from a cluster C_i, if there exists at least one path:

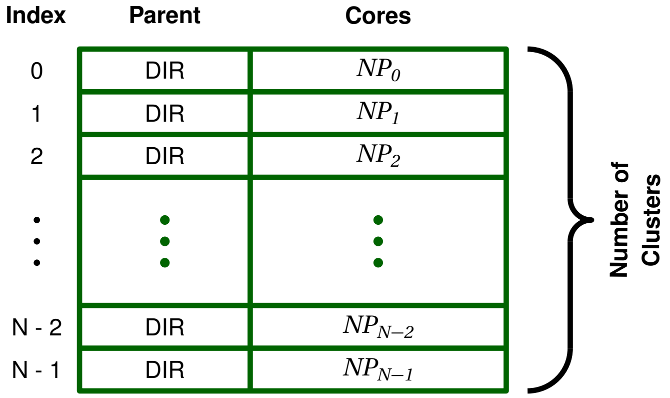
$$C_i, C_{i+1}, C_{i+2}, \ldots, C_{i+n}, C_j \mid C_{i+1} \in NC_i \wedge C_{i+1} \in NC_{i+2} \wedge \ldots \wedge C_j \in NC_{i+n}$$

In can be concluded from Property 4.2.1, Property 4.4.2 and Definition 4.4.1 that, if a cluster C_j is **reachable** from a cluster C_i, then C_j is **reachable** from the cluster C_i. Therefore, the FFST's root can reach reliably any node in the tree, and inversely, any node in the tree can reach reliably the FFST's root.

4.4.1 FFST's Data Structure

The FFST infrastructure is implemented as a distributed data structure. Each local leader i has a local instance of this structure in its local memory, and it is filled during the FFST's construction.

This distributed data structure of a local leader i is called $TREE_i$. Figure 4.8 shows an instance of this structure. It is implemented as an array whose number of entries equals the number of clusters in the mesh. Each entry has two fields: **Parent** and **Cores**. The **Parent** field indicates the parent of the node (NORTH, SOUTH, EAST, WEST). Additionally, the **Parent** field can contain the values ROOT or NULL to indicate that the node is the FFST's root, or does not belong to the $TREE_i$ array, respectively. The **Cores** field contains the list of functional cores in the corresponding cluster. This list is the NP_i list defined in Property 4.1.2. Finally, each entry's index is defined as $index(x, y) = x * YSIZE + y$, where x and y are the coordinates of the target cluster.



Index	Parent	Cores
0	DIR	NP_0
1	DIR	NP_1
2	DIR	NP_2
⋮	⋮	⋮
N - 2	DIR	NP_{N-2}
N - 1	DIR	NP_{N-1}

DIR ∈ { NULL, ROOT, NORTH, SOUTH, EAST, WEST }

Figure 4.8 – FFST's Data Structure

A tree can be defined recursively as a root node with pointers to the subtrees of its children. These subtrees are mutually exclusive because, by definition, in a tree there is a unique path between two nodes. The FFST is defined this way: each distributed $TREE_i$ does not contain all the nodes in the FFST. Instead, a $TREE_i$ array represents the subtree whose root node is the local leader i. This subtree contains the node i and the nodes in the subtrees of its children. As a consequence, only the $TREE_r$ array of the global leader (r) contains the complete description of the FFST. This is stated in the following definitions:

Definition 4.4.2. The $TREE_i$ of a local leader i is defined as the **merging** of the *TREE* arrays of its children and the node i.

$$TREE_i = \{\{\,i\,\} \cup \bigcup_n TREE_n \mid n \text{ is child of } i\} \tag{4.2}$$

Definition 4.4.3. The **merging** operation $\bigcup$, between two or more *TREE* arrays, is equivalent to the union operator in set theory: the result is an array with all distinct elements in the input *TREE* arrays.

Definition 4.4.4. Let r be the processor's global leader, the $TREE_r$ array contains all the functional clusters reachable from r. This means that $TREE_r \equiv$ FFST.

Figure 4.10 shows the $TREE_r$ array for the example FFST in Figure 4.9. This array belongs to the processor's global leader in the cluster **(0,0)**. There is a faulty router in cluster **(2,1)**. This cluster does not belong to the FFST, and therefore it has the value NULL. In Figure 4.10, the cluster **(0,0)** has the value ROOT because it is the FFST's root. Nodes **(0,1)** and **(1,0)** have the values SOUTH and WEST, respectively, because their parent cluster is **(0,0)**.

As defined above, the FFST's data structure stores uniquely the parent of each node. Therefore, if a local leader wants to know its children, then it must search in its $TREE_i$ array the neighbor nodes entries which has the **Parent** field pointing to itself.

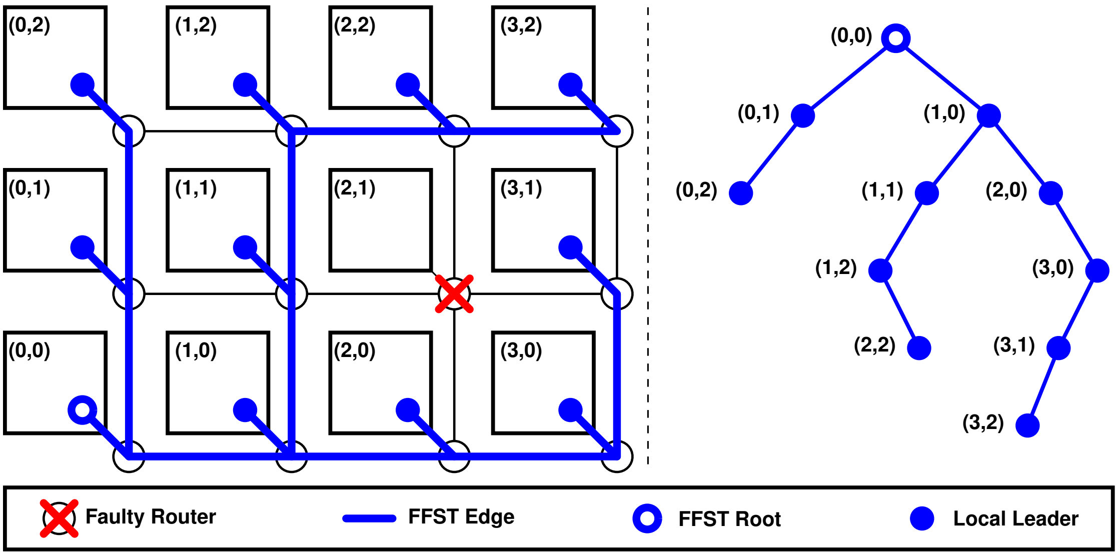


Figure 4.9 – Example of a FFST and its Logical Representation in a 4 × 3 Mesh With a Faulty Router

Index	Parent	Cores
(0,0) = 0	ROOT	$\{P_0, P_1, P_2, P_3\}$
(0,1) = 1	SOUTH	$\{P_0, P_1, P_2, P_3\}$
(0,2) = 2	SOUTH	$\{P_0, P_1, P_2, P_3\}$
(1,0) = 3	WEST	$\{P_0, P_1, P_2, P_3\}$
(1,1) = 4	SOUTH	$\{P_0, P_1, P_2, P_3\}$
(1,2) = 5	SOUTH	$\{P_0, P_1, P_2, P_3\}$
(2,0) = 6	WEST	$\{P_0, P_1, P_2, P_3\}$
(2,1) = 7	NULL	$\emptyset$
(2,2) = 8	WEST	$\{P_0, P_1, P_2, P_3\}$
(3,0) = 9	WEST	$\{P_0, P_1, P_2, P_3\}$
(3,1) = 10	SOUTH	$\{P_0, P_1, P_2, P_3\}$
(3,2) = 11	WEST	$\{P_0, P_1, P_2, P_3\}$

Figure 4.10 – Example $TREE_r$ Array of the Global Leader

4.4.2 FFST's Construction Algorithm

This section explains the algorithm to build the FFST software-based infrastructure. At the end of this algorithm, among all the candidate clusters, one becomes the global leader of the processor. This global leader is the FFST's root, and all the nodes reachable from this global leader are the nodes of this tree as stated in Property 4.4.2. The following property must be also respected:

Property 4.4.3. *At the end of the FFST's construction algorithm, only one local leader (representing a functional IO cluster) is elected global leader.*

The distributed firmware implements, in the local memory of each functional cluster, one input software mailbox for each functional neighbor cluster in its NC_i list.

Figure 4.11 shows a pair of these mailboxes implemented by two neighbor clusters. For a mesh topology, there are at most four mailboxes in a cluster. At the end of the FFST's construction algorithm, one mailbox pair implements one FFST's edge.

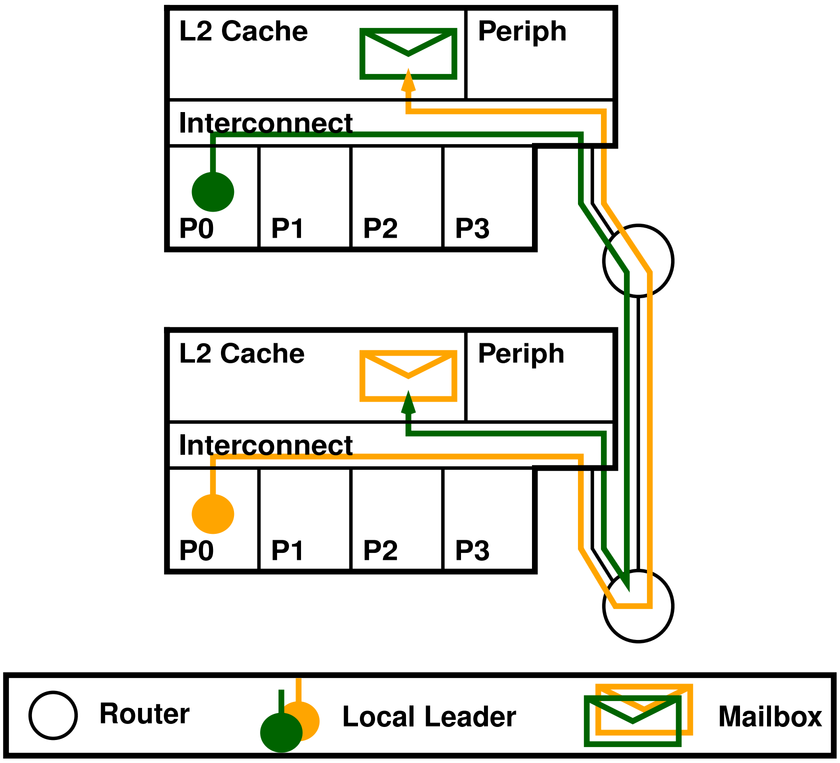


Figure 4.11 – Software Mailboxes Between Two Neighbor Clusters

During this construction algorithm, the candidates to become global leader try to build each a tree in parallel. Each candidate tries to join to its tree all reachable functional clusters. This joining process is performed by means of message propagation. Three kind of messages are used during the FFST's construction: *JOIN*, *ACCEPT* and *DONE*.

- The *JOIN* message is a request from a candidate to another node to join its tree. This message contains the ID of the triggering candidate, and it is broadcast by each receiving node to all its functional neighbors but the sender.
- The *ACCEPT* message is an acknowledgement to a *JOIN* message. It is issued by a node when it accepts (with all its subtree) to join a tree. This message contains the ID of the acknowledged *JOIN* message.
- The *DONE* message is issued by the elected global leader when the FFST's construction completes. It signals all the nodes that the FFST's construction process is finished.

There are two difficulties related to the parallel and asynchronous execution of this algorithm:

1. A node can receive more than one *JOIN* message concerning the same candidate. As the *JOIN* messages are broadcast by each receiving node, if a node

has more than one functional neighbor, it can receive several identical *JOIN* messages from each of them. In that case it handles the first one and discards the others.

2. A node can receive more than one *JOIN* message sent by different candidates. This happens when a node is reachable by several candidates. In this case, when the FFST's construction algorithm completes, the node will belong to the tree of the candidate with the highest priority.

The remainder of this section describes the execution of the FFST construction algorithm.

The local leaders which are not candidates to become global leader start to poll the mailboxes waiting for a message from one neighbor. However, the candidates to become global leader, start the algorithm broadcasting a *JOIN* message to all clusters in their NC_i list. In TSAR there are two candidates to become global leader: cluster **(0,0)** and cluster **(XSIZE-1,YSIZE-1)**. These messages are written into the software mailboxes previously described.

The entire process is illustrated in Figure 4.12. This figure shows a possible execution of the algorithm to build the FFST in Figure 4.9. In this example, the two IO clusters **(0,0)** and **(3,2)** try to build each a tree, while the cluster **(2,1)** is faulty.

When a local leader receives a *JOIN* message, it set its own entry in its local $TREE_i$ structure: it sets the **Cores** field with its NP_i list, and sets the **Parent** field with the sender cluster's direction. Thus, this establishes a FFST's edge between the receiving (child) and sender (parent) clusters. Next, it propagates the *JOIN* message to all its functional neighbor clusters but the sender (see ② in Figure 4.12). This is recursively done to flood the mesh, and the message is received by all the functional clusters reachable from the triggering candidate. Additionally, when a local leader receives a *JOIN* message, it saves the ID of the triggering candidate. The *ACCEPT* message is not sent immediately. This *ACCEPT* message is sent only when all the acknowledgements for the broadcast *JOIN* message are received.

This is a parallel procedure: all the candidates to become global leader broadcast *JOIN* messages to build a tree. However, we want only one tree to become the processor's FFST, and only one of these candidates to become the global leader. To do this, the FFST's construction algorithm assigns a static priority to each candidate. The candidates with lower IDs have higher priorities. When a local leader i receives a *JOIN* message from a candidate n, it checks first if it belongs already to a tree. If i belongs to a tree m, then it checks the priorities. If $ID(n) < ID(m)$, then i changes to the tree n: it resets its $TREE_i$ array, re-adds its entry with the new parent, and broadcasts the new *JOIN* message to its neighbors but the sender (see ③, ④ and ⑤ in Figure 4.12). Otherwise, if $ID(n) > ID(m)$, then i ignores the *JOIN* message from n.

When a local leader propagates a *JOIN* message to all its functional neighbor clusters (but the sender), and waits an acknowledgement for each one. The acknowledgement can be of two kinds: an *ACCEPT* message or a *JOIN* message with the same triggering candidate. As explained before, a node may receive a *JOIN* message

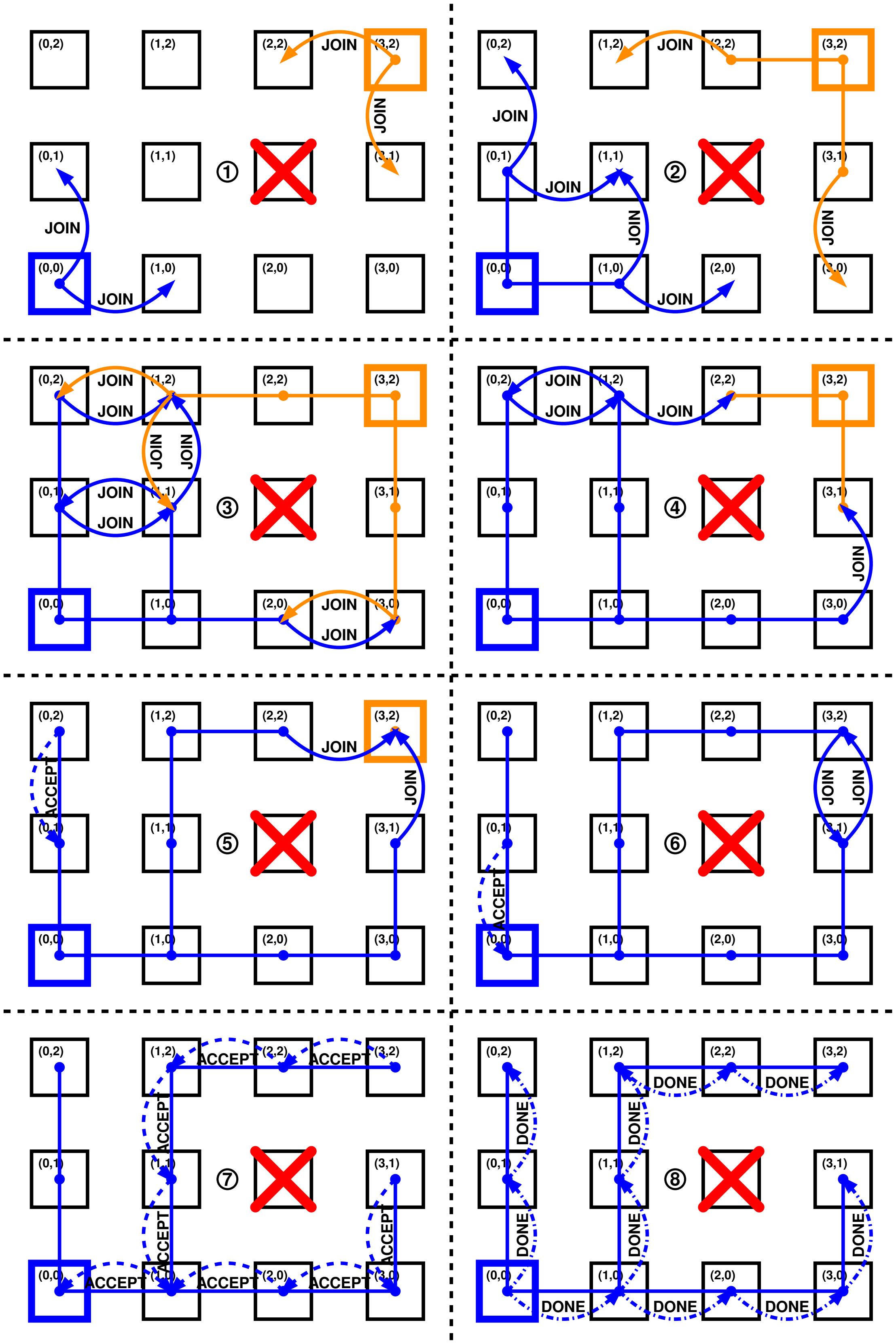


Figure 4.12 – FFST Construction Algorithm Example

from the same candidate more than once (e.g. see ③, ④ and ⑤ in Figure 4.12). In that case, the redundant *JOIN* messages mean that the sender node already belongs to another branch of the same tree.

The *ACCEPT* message is sent from a local leader to its parent when it has received all the acknowledgements for all previously sent *JOIN* messages (see ⑤, ⑥ and ⑦ in Figure 4.12). When a local leader i receives an *ACCEPT* message from a neighbor n, it means that n has joined the tree and accepted i as parent. In this case, the local leader i merges, using Definition 4.4.3, the $TREE_n$ array of n with its own $TREE_i$ array. To perform this merge, the local leader i accesses the local memory of n, and it copies into its $TREE_i$ array the entries in the $TREE_n$ array.

When a local leader i receives all the acknowledgments for a previously sent *JOIN* message, it has in its local $TREE_i$ structure the representation of the subtree for which it is the local root. And in particular, if a candidate to become global leader i receives all acknowledgements, then it has in its $TREE_i$ structure a tree with all the nodes that it can reach. When all the candidates are mutually reachable, at the end of the algorithm, the global leader is the candidate with the smallest index because all reachable nodes switched to the smallest index candidate.

Special Case: Partitioned Mesh

When there are too many black-holes in the mesh (deactivated faulty routers or faulty clusters), the graph may be partitioned in several non-connected sub-graphs which cannot communicate with each other.

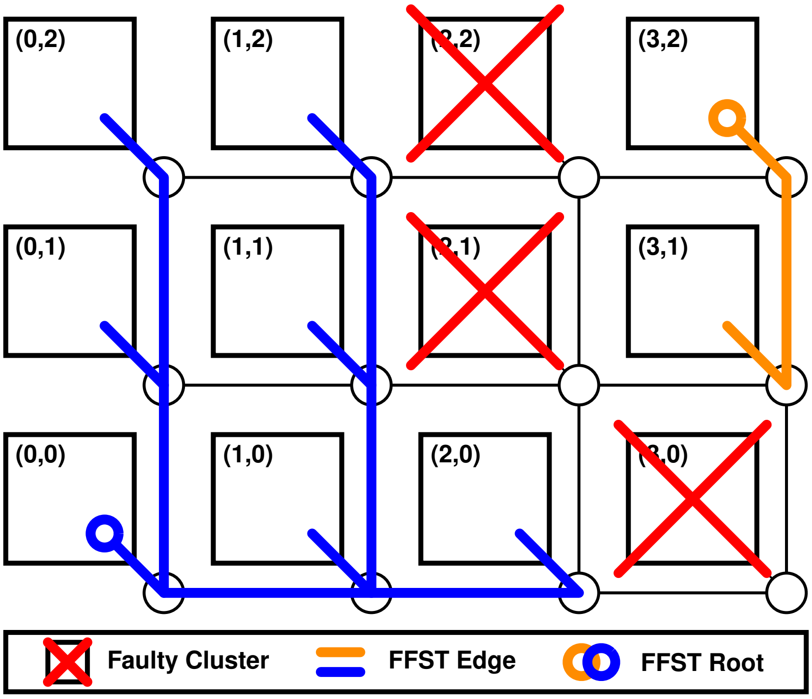


Figure 4.13 – FFST Example With Two Partitions

If there are two candidates to become global leader in different sub-graphs, then there will be two different trees at the end of the FFST's construction algorithm. An example is shown in Figure 4.13. In this example, as there is no path (as defined in Definition 4.4.1) connecting both candidates, then the *JOIN* messages from each candidate never find each other, and each candidate builds a different tree.

In such cases, the distributed recovery firmware must choose the tree with the greatest number of clusters, and deactivate the others. However, as the trees belong to different sub-graphs, a global consensus is not possible. For this reason, the distributed firmware uses a decentralized method to choose the FFST. When a candidate to become global leader finishes the construction of its tree, it counts the number of reached nodes. If this number is greater than $N/2$ (where N is the total number of clusters in the mesh), then it becomes the global leader, and its tree becomes the FFST. In contrast, if the number of reached nodes is less or equal than $N/2$, then the candidate self-deactivates. In the example of Figure 4.13, the local leader of cluster **(0,0)** becomes the global leader, and the local leader of cluster **(3,2)** self-deactivates and send a message to its descendants to signal them that they should deactivate. If no tree has more than $N/2$ clusters, the entire processor is considered faulty. This leads to the following property:

Property 4.4.4. *The processor can recover from failures when there is at least one functional IO cluster, and one of these functional IO clusters can reach more than $N/2$ clusters (where N is the total number of clusters in the processor).*

When a candidate becomes global leader, it broadcasts a *DONE* message through the FFST (see ⑧ in Figure 4.12) to inform its descendants that the FFST is built, and they can all enter the next phase of the distributed recovery firmware.

Proof of the FFST's Construction Algorithm Properties

First the Property 4.4.1 is proved. In Section 4.2.1, it was proved that at the end of the intercluster phase, each local leader i has a NC_i list containing all its functional clusters with which a full-duplex communication is possible. Therefore, if there are two clusters A and B, and $B \in NC_A$, then $A \in NC_B$ to effectively support a bidirectional communication. During the FFST's construction algorithm, the *JOIN* messages to build the tree are propagated by each local leader to the functional neighbor clusters in their *NC* list. As the FFST's edges are established when a cluster receives a *JOIN* message, then two clusters A and B connected by an FFST's edge, will always respect that $A \in NC_B$, and $B \in NC_A$.

Property 4.4.2 is proved as follows. As explain in Section 4.4.2, the *JOIN* messages are broadcast by all local leaders to flood the mesh. If a cluster C_m is reachable by the global leader in C_r, then there exists a path (as stated in Definition 4.4.1) consisting in a sequence of neighbor clusters from C_r to C_m. Therefore, the *JOIN* message will reach C_m, and C_m will join the FFST.

Finally, to prove the Property 4.4.3, as explained in Section 4.4.2, each candidate to become global leader has a priority. When more than one candidate is reachable from another candidate, the *JOIN* message of the candidate with the highest priority will prevail, and all reachable clusters will belong to its tree (including the other candidates that join this tree and stop their attempt to build a tree of their own). In the case where the mesh is partitioned, when a tree has more than $N/2$ clusters, it becomes the FFST, and as each sub-graph contains an exclusive set of clusters, the other sub-graphs have necessarily less than $N/2$ clusters and self-deactivate.

4.5 Map of Operational Resources

This section describes how the distributed recovery firmware uses the FFST to build a global map of the operational computational and communication resources of the many-core processor.

The new global routing function implemented after the NoC reconfiguration is computed based on the global map of the operational communication resources built during the distributed information gathering. The remainder of this section details the distributed information gathering. NoC reconfiguration will be explained in Chapter 5.

4.5.1 Distributed Information Gathering

In the previous stages, four distributed procedures were executed: the first is the NoC BIST (Section 3.2) where each NoC router self-tests and self-deactivates if it detects a fault; the second is the intracluster test phase (Section 4.1) where all the cores determine if they can be declared functional; the third is the intercluster test phase (Section 4.2) where the functional full-duplex communication channels between neighbor clusters are identified; the fourth is the FFST's construction and global leader's election (Section 4.4). All this diagnostic information is distributed, and therefore there is not a global status of the processor. The FFST's global leader gathers all this distributed information, and builds a global map of the processor's operational hardware. The operational hardware of this global map can be classified in two categories: (1) operational computational resources (clusters), and (2) operational communication resources (NoC routers).

The map of the computational resources is contained in the FFST's root node. The detailed information about the functional cores at each cluster is known, because during the FFST's construction, the local leaders fill the **Cores** field of the $TREE_i$ structure. And the functional memory banks are known, because in a cluster self-declared functional, the memory bank is functional. Therefore, the $TREE_r$ structure of the root cluster already contains a complete description of the computational resources.

To complete the global map of the operational hardware, only the map of operational communication resources is missing (accurate location of the functional routers in the NoC). This location procedure is called *Black-Holes Location.*

4.5.2 Black-Holes Location Procedure

Consider two clusters C_A and C_B. When C_A tries to communicate with C_B during the intercluster phase (Section 4.2), the test can fail because: (1) there are one or more faulty router in one network between C_A and C_B, or (2) C_B is faulty. Then, the intercluster test phase cannot decide if the communication resources are faulty.

We propose another software procedure, which relies on the FFST, to locate the faulty routers. This procedure is executed after the FFST's construction, and it is triggered by the global leader. The FFST's root cluster broadcast a *BLACK-HOLE LOCATION* message through the FFST to all nodes. When a node receives this message, it propagates the message to its children and executes the software procedure explained below.

When a node finishes the procedure, it waits for the acknowledgements of all children before sending an acknowledgement to its parent. If the global leader receives the acknowledgement from all its children, the black-hole location procedure is finished.

When a local leader receives the *BLACK-HOLE LOCATION* message, it tests the communication with all the clusters in the mesh (functional or not) through all NoCs. In order to test the *DIRECT* networks (*CMD* and *RSP*), every local leader tries to read the memory in all the clusters. And to test the *COHERENCE* networks (*P2M, M2P* and *CLACK*), every local leader uses the mechanism explained in Section 4.3 to access the L2 cache in all clusters.

If there is one or more faulty routers (black-holes) in the path between the initiator local leader and its target, then the watchdog timer of the local leader triggers an exception allowing the software to diagnose the path. If, on the contrary, a local leader tests successfully another cluster, this local leader *tags* all routers in the path as functional. Tagging a router means to register it in a table, stored in the local memory of the cluster. This table is called Locally Discovered Routers Table (LDRT). There is one LDRT per NoC, and the number of entries of each table equals the number of clusters in the mesh. This table contains, at each entry, the status of a router in the associated NoC. This status is encoded in one bit: (0) faulty or (1) functional. In TSAR, each local leader fills four tables (*CMD, RSP* and *P2M* NoCs have its own table, and the *M2P* and *CLACK* NoCs share one). As the maximum mesh's size is $16 \times 16 = 256$ clusters, each table occupies 256 bits of memory (1 Kbit is the total for the four tables).

During the black-hole location procedure, the NoCs are not reconfigured, and all routers implement the X-first routing algorithm. Then, when a local leader tests another cluster, the transactions' packets follow a deterministic X-first path, and the local leaders use Algorithm 4.4 to tag the routers. This algorithm receives as parameters the coordinates of the source and target clusters, and a LDRT. A transaction consists in a command and then a response or acknowledgement, thus this algorithm is used twice for each transaction: the first time to tag the routers used by the command packet, and the second time to tag the routers used by the response or acknowledgement.

Algorithm 4.4: Tag X-First Path Algorithm

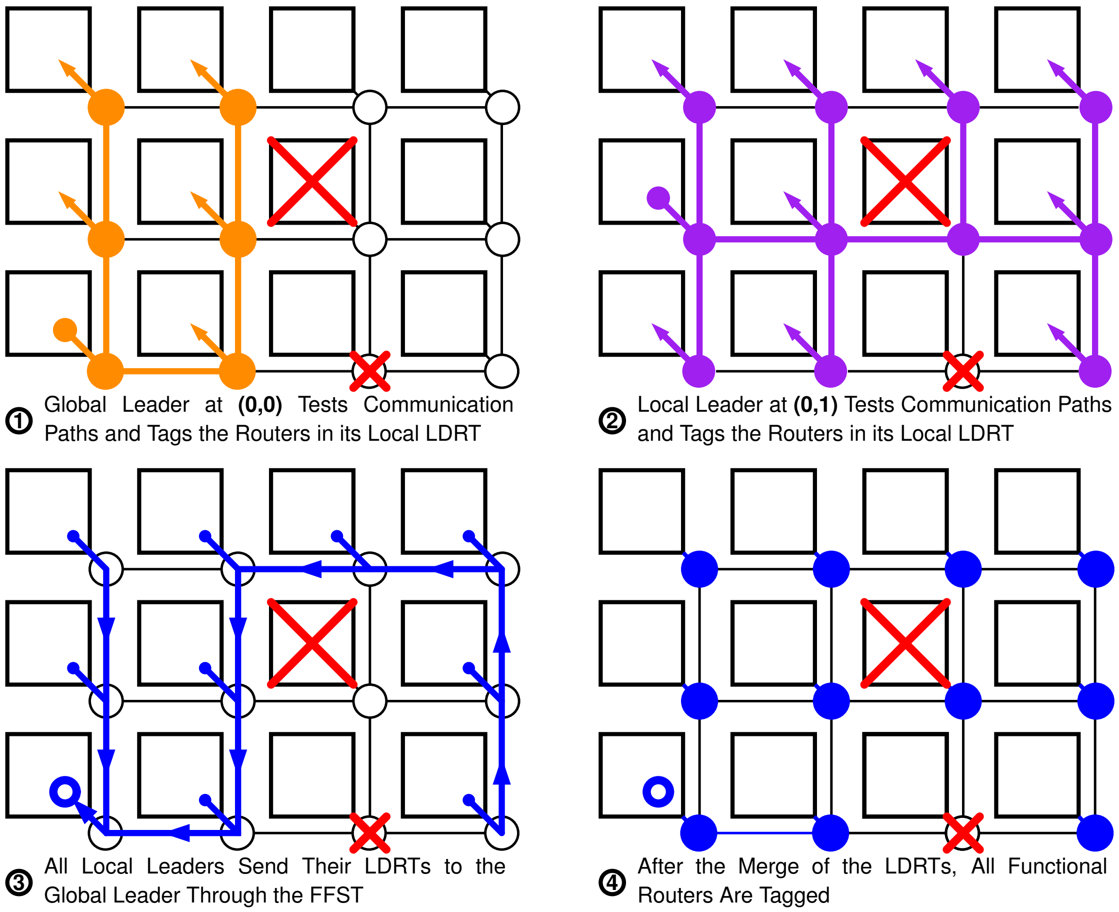
```
TagPath()
Input: $LDRT$: a table with the status of the routers in a NoC
Input: $I_X, I_Y$: the $X$ and $Y$ coordinates of the initiator cluster
Input: $T_X, T_Y$: the $X$ and $Y$ coordinates of the target cluster
begin
    /* Tag the router on the initiator coordinates                    */
    $LDRT[I_X][I_Y] = 1$

    /* Tag the routers on the $X$ direction                           */
    $X_{min} = \min(I_X, T_X)$
    $X_{max} = \max(I_X, T_X)$
    for $x = X_{min}$ to $X_{max}$ do
        if $x \neq I_X$ then $LDRT[x][I_Y] = 1$
    end

    /* Tag the routers on the $Y$ direction                           */
    $Y_{min} = \min(I_Y, T_Y)$
    $Y_{max} = \max(I_Y, T_Y)$
    for $y = Y_{min}$ to $Y_{max}$ do
        if $y \neq I_Y$ then $LDRT[T_X][y] = 1$
    end
end
```

For example, when testing the *CMD & RSP* networks, if a local leader A in a cluster (X_A, Y_A) tests a cluster (X_B, Y_B), and this test succeeds, then A calls:

- TagPath($LDRT_{CMD}$, (X_A, Y_A), (X_B, Y_B)) to tag the *CMD* routers.
- TagPath($LDRT_{RSP}$, (X_B, Y_B), (X_A, Y_A)) to tag the *RSP* routers.

Note that for the *RSP* network, the initiator and target coordinates are inverted because the target is the sender of the response. Analogously, during the test of the *coherence* networks, the local leaders A uses the TagPath function on the $LDRT_{P2M}$ and the $LDRT_{M2P\,\&\,CLACK}$ to tag the routers in those networks when the test succeeds.

Figure 4.14 shows an execution example of the black-hole location procedure in a 4 × 3 mesh with a faulty cluster at **(2,1)** and a faulty router at **(2,0)**. This example only represents one NoC for the sake of simplicity. After a local leader finishes the test of all other clusters, it has only the status of all communication paths that it can test (see ① and ②). Therefore, in order to build a global map of the communication resources, all the LDRTs in all clusters should be merged to have one single LDRT per NoC. This merge is made recursively (see ③). After a node finishes the local execution of the black-hole procedure, it waits an acknowledgement of its children signalling that they finish also the procedure. When they do, each local leader merges its LDRTs with those of its children, and then it sends an acknowledgement to its parent. Eventually the global leader receives the acknowledgement from its children, and it merges also its LDRTs with the ones of its children.



① Global Leader at **(0,0)** Tests Communication Paths and Tags the Routers in its Local LDRT

② Local Leader at **(0,1)** Tests Communication Paths and Tags the Routers in its Local LDRT

③ All Local Leaders Send Their LDRTs to the Global Leader Through the FFST

④ After the Merge of the LDRTs, All Functional Routers Are Tagged

Figure 4.14 – Black-Hole Location Procedure

At the end, the resulting $LDRT_{CMD}$, $LDRT_{RSP}$, $LDRT_{P2M}$ and $LDRT_{M2P\,\&\,CLACK}$ in the global leader's cluster, represent the global map of the communication resources (see ④).

4.6 Conclusion

This chapter explained the proposed software-based solution for locating faulty cores, faulty memory banks and faulty NoC routers in a shared memory many-core architecture. This solution uses a ROM distributed in all clusters which contains the recovery firmware executed at each processor's reset or power-on by all the cores.

The cores self-test, test their cluster, build the FFST software-based communication infrastructure, and finally use this FFST to centralize the distributed diagnostic information and build a global map of the operational hardware. The global map considers the computational resources (processor cores and memories) and the communication resources (routers of the different NoCs).

The global map of computation resources is the one passed to the OS after it is loaded, so it can self-adapt to the actual hardware.

The global map of communication resources is used to compute a new global routing function, which allows bypassing the faulty routers. To implement this new routing function the NoCs routers need to be reconfigured. This work proposes a software mechanism to reconfigure the routers which is based on the use of the FFST. This solution is presented in Chapter 5.

Though the proposed solution is based on the execution of a distributed firmware by the cores, four hardware mechanisms are required by this solution. A watchdog timer at each L1 cache, a mechanism allowing the software to directly test the coherence networks between cache controllers, a mechanism to use the L2 cache controller as a local RAM, and the gateway hardware barrier mechanism to allow disconnecting a cluster from the global NoC.

The entire solution has been validated and evaluated on a cycle-accurate and bit-accurate virtual prototype of the TSAR architecture. The performance evaluation and the hardware cost will be presented in Chapter 6.

Chapter 5

NoC Reconfiguration

Contents

5.1 Introduction

When there are holes in the NoC (faulty routers), the communication between clusters is broken, and a reconfiguration mechanism is needed to modify the global routing function, and repair the intercluster communication. This new global routing function needs to bypass the holes in the NoC, so that the remaining operational cores may communicate to each other.

When starting the NoC reconfiguration phase, the faulty routers were deactivated at the power-on by the distributed hardware BIST and behave as black-holes. Then, these holes have been located using the distributed software-based location procedure presented in Chapter 4.

Additionally, a software-based communication infrastructure, called FFST, has been built. This FFST allows the global leader (a core in an IO cluster) to communicate reliably with the other operational cores in the processor, by means of software point-to-point full-duplex communication channels between operational neighbor clusters. This FFST was used, in particular, to centralize the information gathered during the distributed software-based location procedure into the cluster of the global leader, and build a global map of the computational and communication operational resources.

Even if the FFST provides a communication infrastructure allowing the intercore communication, the NoC still needs to be reconfigured. The FFST is only a temporary software infrastructure with very low communication bandwidth. Moreover, the cores behave as software routers which relay messages in a neighbor-to-neighbor basis incurring important communication latencies.

In order to tolerate holes in the NoC, we choose to use the reconfigurable routing algorithm proposed by Zhang, Greiner, and Taktak [15] (explained in Section 2.2.4). In this algorithm, the routers in the contour of the holes need to be reconfigured in order to modify the routing algorithm and repair the communication. This algorithm is called recovery routing algorithm.

The recovery routing algorithm needs the NoC holes to be located, and requires a reliable communication infrastructure to reconfigure the routers in their contour. We propose a fully software-based solution for both problems in order to limit the hardware overhead of our complete fault-tolerance strategy.

We propose to use the global map of communication resources, built during the software-based location procedure, to know the location of holes; and we propose to use the FFST as the communication infrastructure to reconfigure the routers in the contour of the holes.

The use of the FFST avoids the integration of a specific hardware reconfiguration network by reusing the existing communication resources. As these communication resources may be partially defective, the FFST uses only the resources previously tested and declared functional during the distributed recovery firmware execution. This allows a reliable reconfiguration of the NoC.

At this phase, most of the software procedure allowing the NoC reconfiguration are executed by the global leader of the FFST. As the global leader has direct access to the IO peripherals, the procedures which are exclusively executed by the global leader are stored in an external mass storage device in order to reduce the size of the distributed ROMs.

In the following, Section 5.2 describes the NoC reconfiguration procedure. Then, Section 5.3 describes a new mechanism to allow the reallocation of the memory segment of a deactivated cluster to one of its neighbor, and finally, Section 5.4 describes a new mechanism to support broadcast when there are holes in the NoC.

5.2 Reconfiguration Procedure

The NoC reconfiguration procedure begins after the black-hole location procedure (Section 4.5.2), when the global leader has built the global map of the communication resources. Therefore, the global leader knows the location of the holes (if any), in the five NoCs of the TSAR architecture (CMD, RSP, M2P, P2M & CLACK).

The recovery firmware is responsible for the NoC reconfiguration. The global leader sends reconfiguration commands, through the FFST, to the clusters in the contour of the holes. Then, the local leaders in these clusters reconfigure the local routers by writing in their reconfiguration register. These reconfiguration commands are implemented as messages emitted by the global leader, and propagated through the FFST to their destination.

During this process, the local leaders of the clusters in the FFST behave as software routers. When the global leader needs to send a reconfiguration command, it searches in its $TREE_r$ data structure (detailed in Chapter 4) to which child's subtree belongs the target cluster, and sends the reconfiguration message to that child. Then, recursively, each local leader, receiving a message, forwards this message to one of its children (based on its local $TREE_l$ data structure) until the message reaches its destination. At this point, the local leader in the target cluster reconfigures the target local NoC router, and sends an acknowledgement message to the global leader.

In TSAR, the writing in the reconfiguration register of the NoC routers is performed through the XICU, because the NoC routers are not directly addressable by the software. As there are five independent NoCs, there are five memory-mapped reconfiguration registers in the XICU allowing to reconfigure independently each of the five local routers.

At the end of the NoC BIST executed at power-on, the routers that passed the hardware test, are configured with the NORMAL value in their reconfiguration register (use the default X-first routing algorithm).

The reconfiguration commands sent by the global leader contains the coordinates of the target cluster, an index corresponding to one of the five NoCs (0: CMD, 1: RSP, 2: M2P, 3: P2M, and 4: CLACK), and the relative position of the target cluster with respect to the hole (NE_OF_X, N_OF_X, NW_OF_X, E_OF_X, W_OF_X, SE_OF_X, S_OF_X, SW_OF_X). When the target cluster receives a reconfiguration message, it writes the configuration in the local XICU, accordingly to the message information.

The global leader sends the reconfiguration messages sequentially to all the clusters in the contour: it sends a message to a cluster, and waits for its acknowledgement before sending another reconfiguration message to the next cluster in the contour. When the global leader reconfigures the contour of all holes in the different NoCs, the NoC reconfiguration phase is finished. At this point, the global leader broadcast a specific message to all the local leaders in the FFST, to signal that they must pass to the OS loading phase.

After the NoC reconfiguration phase, the new implemented global routing algorithm bypasses the holes in the NoC. Therefore, the FFST is not needed anymore for the intercluster communication.

As explained in Chapter 4, when there are one or more faulty routers in a cluster, the entire cluster is deactivated and behaves as a black-hole. Therefore, because of the physical distribution of the address space, the memory segment of deactivated clusters cannot be accessed by the OS. In Section 5.3, we propose a mechanism to reallocate the memory segment of a deactivated cluster to one of its neighbor clusters, so as to reduce the memory loss incurred by faults.

5.2.1 Supported NoC Faulty Topologies

In order to support a faulty router in a NoC, the following property must be met:

Property 5.2.1. *All the clusters in the contour of a hole must be functional, and belong to the FFST.*

This condition is needed in order to allow the reconfiguration of all the routers on the contour of a hole. Of course, all the routers on the contour of a hole need to be also functional, but this is guaranteed by the condition above because a cluster with one or more faulty routers is deactivated and cannot belong to the FFST.

The reconfigurable routing algorithm supports any single-faulty-router topology in the NoC, and the authors have demonstrated that this algorithm guarantees the deadlock and livelock freedom for this kind of topologies (one faulty router).

In TSAR, the five NoCs (CMD, RSP, M2P, P2M & CLACK) are physically independent, and each can be reconfigured in order to support holes. Therefore, our fault-tolerance strategy supports at most one faulty router per NoC.

5.3 Memory Segment Reallocation

As explained in Chapter 1, the TSAR architecture implements a physically distributed address space, where each cluster controls one physical memory segment (typically one gigabyte per cluster), hence, if a cluster is deactivated, the corresponding physical memory becomes unusable for the OS. Figure 5.1 illustrates the distribution of the memory address space on a 4-clusters architecture.

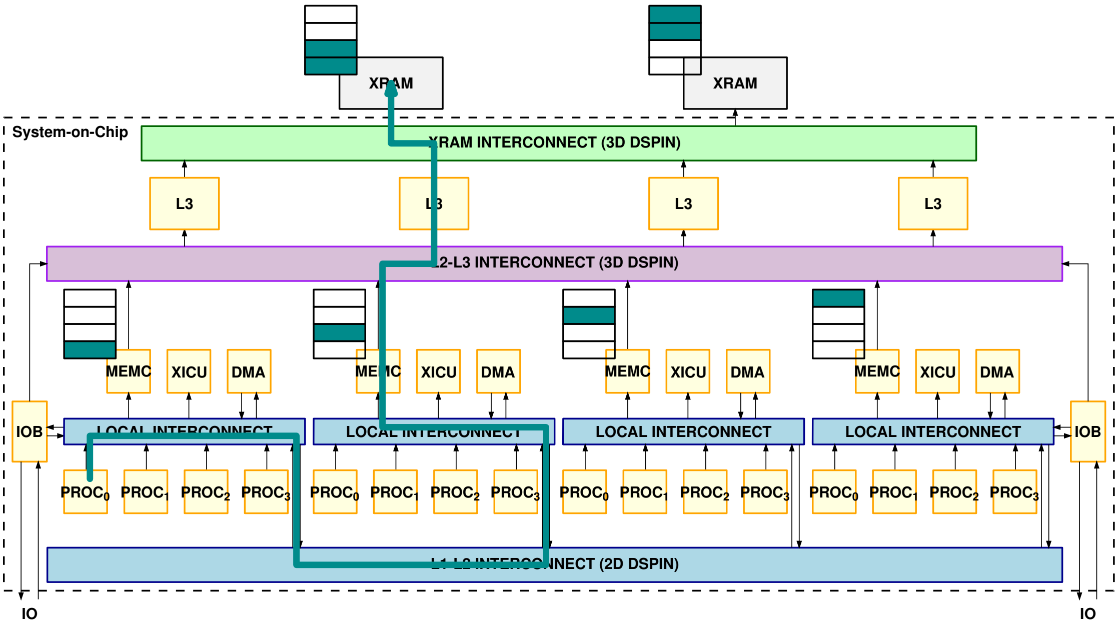


Figure 5.1 – Distribution of the Physical Address Space in TSAR

Fortunately, in TSAR, all clusters can communicate through several physically independent networks (L1-L2, L2-L3 & XRAM). In case of a deactivated cluster, the L2-L3 and XRAM NoCs can be used to reach the memory segment of this cluster.

We propose a reconfiguration of the global routing algorithm to reallocate the physical memory segment of a deactivated cluster to a functional neighbor. An example of such segment reallocation is shown in Figure 5.2.

When a cluster is functional, its associated physical memory segment is accessed through its local L2 cache controller. However, if this cluster is deactivated, after the reallocation, its physical memory segment can be accessed through the L2 cache controller of one of its neighbors.

5.3.1 Implementation

The segment reallocation is based on the same idea of going around a deactivated cluster. The routers in this contour are reconfigured in order to reroute the packets, whose destination is the deactivated cluster, to one of its neighbors.

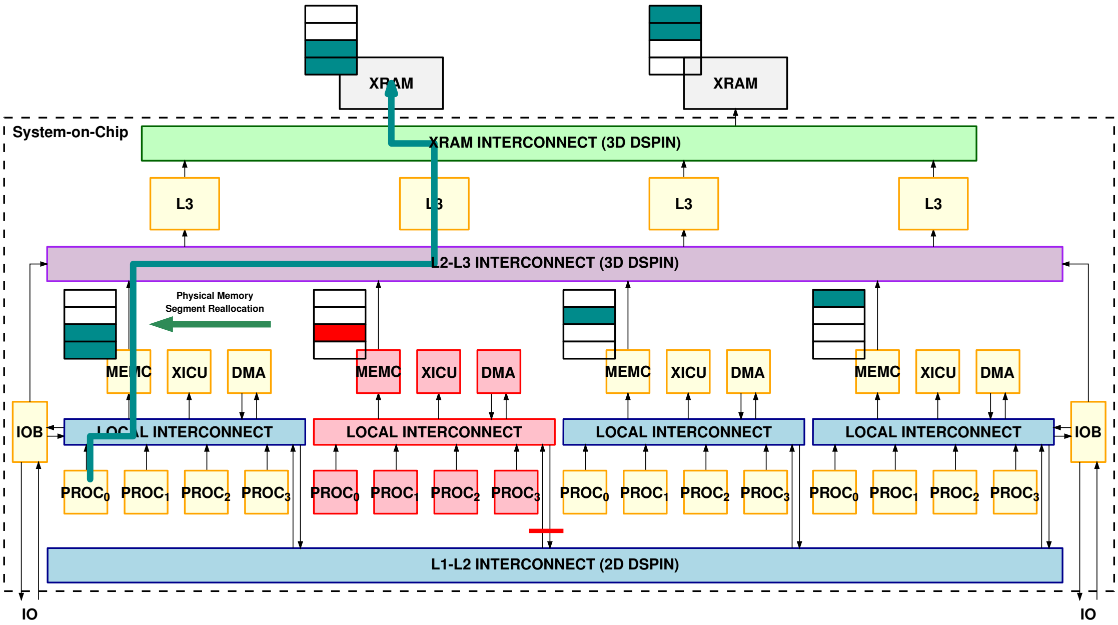


Figure 5.2 – Reallocation of a Physical Memory Segment From a Deactivated Cluster to One of its Neighbors

In order to reduce the hardware cost of the solution, the segment reallocation mechanism reuses the information in the configuration register of the NoC routers for the recovery routing algorithm (position in the contour: N_OF_X, NE_OF_X, etc) to determine the coordinates of the deactivated cluster.

It requires an additional 4-bits register with two fields: the direction to which a packet should be forwarded when the destination is the deactivated cluster, called *Reallocation Direction*; and a bit, called *Blackhole Bypass*, which is explained below. Both informations need also to be written by the recovery firmware during the NoC reconfiguration. Figure 5.3 shows the complete reconfiguration register of the routers in the NoC.

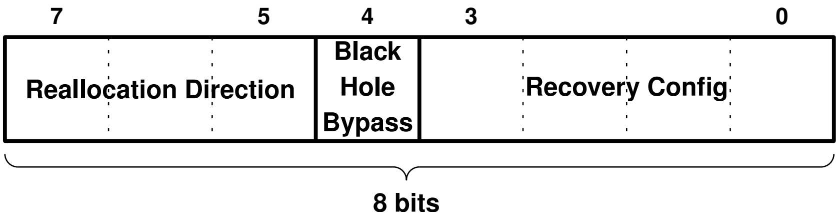


Figure 5.3 – NoC Routers' Reconfiguration Register

The possible values for the *Reallocation Direction* are: *NORTH, SOUTH, WEST, EAST,* and *LOCAL*. When a router in the contour of a deactivated cluster receives a packet whose destination is this last, it forwards the packet to the north, south, west, east, or to the local cluster, respectively.

The *Blackhole Bypass* bit determines if the recovery routing algorithm is used, or only the segment reallocation mechanism is enabled. As explained in Chapter 4, a cluster may be deactivated because of a faulty router, or because it did not pass its intracluster tests (e.g. all local cores are faulty). If a cluster is deactivated, but its routers are functional (the routers belong to the communication resources global map after the black-hole location procedure), these routers can still be used so the network bandwidth, and latency are not degraded unnecessarily. Therefore, when the bit *Blackhole Bypass* is unset, the routers in the contour of a deactivated cluster only reroute packets whose destination is this deactivated cluster, otherwise, the packets are still routed using the normal X-First routing.

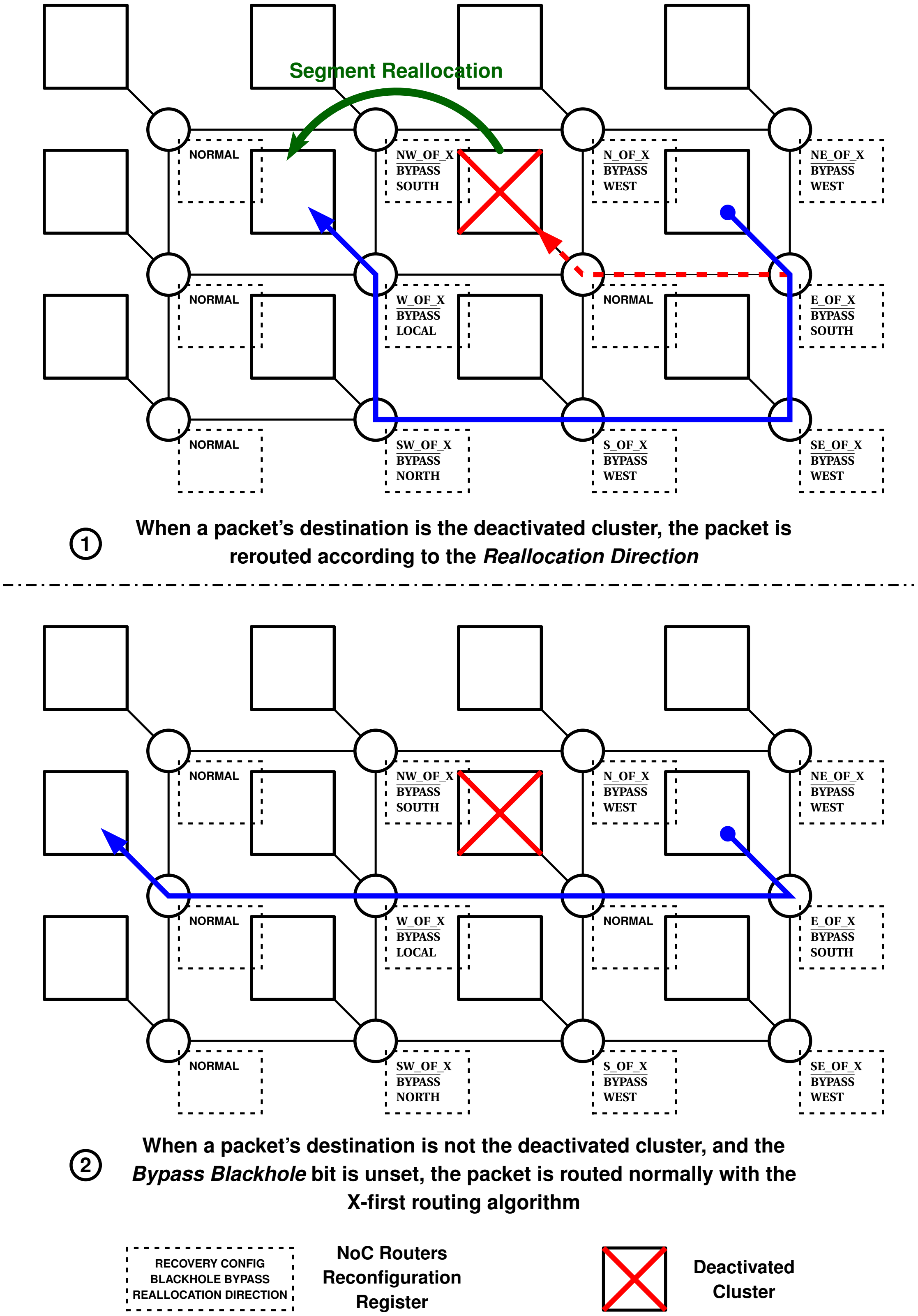
The routers in the contour of a deactivated cluster perform a comparison between the destination coordinates of the incoming packets, and the coordinates of the deactivated cluster. If they match, the packet need to be rerouted. The coordinates of the deactivated cluster are computed locally by the router using the information in its reconfiguration register, and its local coordinates. For example, when its configuration register contains N_OF_X, and its local coordinates are (X_L, Y_L), if it receives a packet whose destination coordinates are $(X_L, Y_L + 1)$, it forwards the packet according to *Reallocation Direction.*

Figure 5.4 ① shows an example of the reconfiguration of a contour to reallocate the segment of a deactivated cluster to the neighbor cluster on its west direction. This figure shows for each router their respective reconfiguration register values. In this example, the *Blackhole Bypass* bit is unset, hence only the packets to the deactivated cluster are rerouted. Figure 5.4 ② shows another example, where the cluster in coordinates (3,1) sends a command to the cluster in coordinates (0,1). As the target cluster is not the deactivated one, and the *Blackhole Bypass* bit is unset, the command packet is routed normally using the X-first routing algorithm. Table 5.1 summarize the behavior of the routers with respect to the configuration values.

Recovery Config	Reallocation Direction	Blackhole Bypass	Routing Decision
NORMAL	-	-	Forward the packets according to the X-first routing algorithm.
≠ NORMAL	DIR	0	If the destination is the deactivated cluster, the packet is forwarded to DIR. Otherwise, the packet is forwarded according to the X-first routing algorithm.
≠ NORMAL	DIR	1	If the destination is the deactivated cluster, the packet is forwarded to DIR. Otherwise, the packet is forwarded according to the recovery algorithm.

Table 5.1 – Routers Routing Decision With Respect to the Reconfiguration Values

To perform the reallocation of a memory segment, not all the NoCs need to be reconfigured: only the CMD & P2M NoCs. The reason is that these networks transmit packets from the cores, and use memory addresses for routing the packets. On the contrary, the RSP, M2P & CLACK networks use cores' identifiers for routing. As all the cores in a deactivated cluster are also deactivated, and cannot communicate with other clusters, it is not possible that a packet on the RSP, M2P & CLACK networks would address one of these cores.



1 When a packet's destination is the deactivated cluster, the packet is rerouted according to the *Reallocation Direction*

2 When a packet's destination is not the deactivated cluster, and the ***Bypass Blackhole*** bit is unset, the packet is routed normally with the X-first routing algorithm

NoC Routers Reconfiguration Register

Deactivated Cluster

Figure 5.4 – Example of a Physical Memory Segment Reallocation With the Cluster (2,1) Deactivated

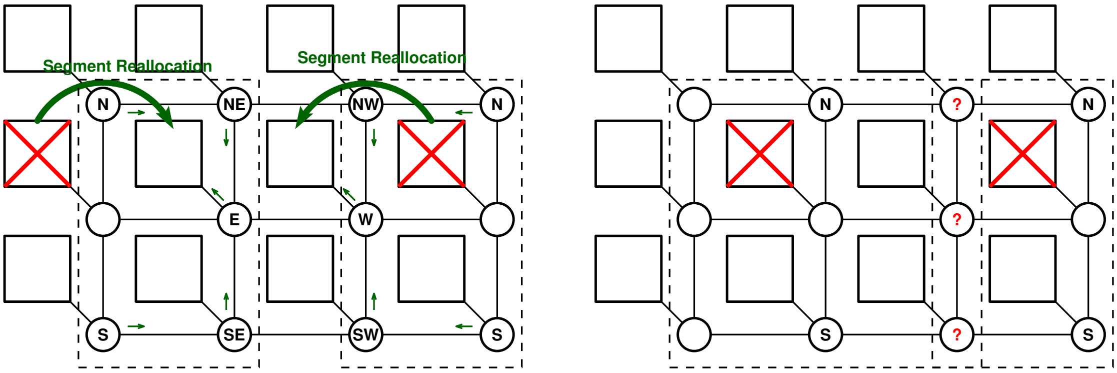
5.3.2 Limitations of the Segment Reallocation Mechanism

The segment reallocation mechanism needs the recovery firmware to reconfigure the routers in the contour of deactivated clusters. Therefore, Property 5.2.1 must be met. Section 5.2.1 explained that only one faulty router per NoC is supported. However, the segment reallocation mechanism supports the reallocation of more than one segment if the following property is also met (in addition to Property 5.2.1):

Property 5.3.1. *None of the clusters in the contour of a deactivated cluster belongs also to the contour of another deactivated cluster (contours are disjoint).*

When Property 5.3.1, or Property 5.2.1 are not met, the segment of the deactivated cluster is not reallocated. The corresponding memory is lost, and the packets that targets this segment are forwarded to the deactivated cluster, and dropped by its gateway hardware barrier mechanism. However, these memory segments are not included in the Device Tree Blob (DTB) structure containing the operational platform resources, and should not be used by the OS.

Figure 5.5 shows one example of a supported scenario where two memory segments can be reallocated, and another example of a non-supported scenario where two memory segments cannot, because their contours are not disjoint.



Supported scenario: the contour of both deactivated clusters are disjoint.

Non-supported scenario: the contour of both deactivated clusters are not disjoint. The routers with the ? symbol are in both contours.

Figure 5.5 – Examples of a Supported and a Non-Supported Scenario for the Physical Memory Segment Reallocation

5.4 Broadcast Support With Holes in the NoC

In the TSAR architecture, all the NoCs implement a unicast communication style where each packet is intended for only one target device. However, the M2P NoC must also support a broadcast communication. In the cache coherence protocol between L1 and L2 caches, a L2 cache can send an invalidation request, which must be delivered to all the L1 caches in the architecture.

The DSPIN NoC (used by all the NoCs in the TSAR architecture), implements a broadcast replication policy, which defines how an incoming broadcast packet on an input port is replicated, and forwarded to the output ports. This broadcast replication policy must guarantee the Property 5.4.1.

Property 5.4.1. *A broadcast packet from an initiator is delivered to all the targets in the network, and each target receives the broadcast packet strictly once.*

In the DSPIN NoC, the broadcast replication policy consists in replicating, and forwarding the packet based on the X-first routing algorithm:

- When the incoming port is WEST (resp. EAST), the packet is replicated and forwarded to the EAST (resp. WEST), NORTH, SOUTH, and LOCAL output ports.
- When the incoming port is NORTH (resp. SOUTH), the packet is replicated and forwarded to the SOUTH (resp. NORTH), and LOCAL output ports.
- When the incoming port is LOCAL, the packet is replicated and forwarded to the EAST, WEST, NORTH, and SOUTH output ports.

This X-first broadcast replication policy inhibits the same turns as the X-first routing algorithm in order to guarantee the deadlock freedom. Additionally, as the X-first routing algorithm guarantees that there exists a unique and deterministic path between an initiator and a target, a target receives a broadcast packet only once. Figure 5.6 shows an example of the broadcasting from the cluster (0, 1), where the NoC replicates and forwards the broadcast packet according to the X-first replication policy.

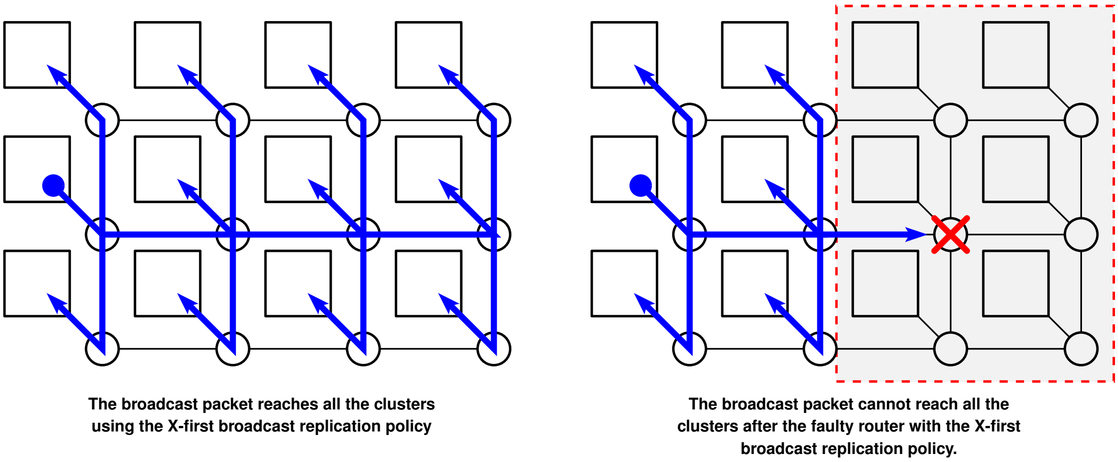

The broadcast packet reaches all the clusters using the X-first broadcast replication policy
The broadcast packet cannot reach all the clusters after the faulty router with the X-first broadcast replication policy.

Figure 5.6 – Example of Packet Broadcasting With the X-First Replication Policy

However, when there are holes in the M2P NoC, the X-first replication policy cannot deliver a packet to all clusters because some paths are broken. Figure 5.6 shows another example where the router in coordinates (2, 1) is faulty, and the X-first replication policy cannot deliver a broadcast packet to all the clusters. Therefore, as for the unicast communication, a new reconfigurable broadcast replication policy is needed in order to bypass the holes, and guarantee the Property 5.4.1.

5.4.1 Recovery Broadcast Replication Policy

We propose a reconfigurable broadcast replication policy in order to support broadcast communications in any single-faulty-router topology. We called this mechanism as recovery broadcast replication policy in the remainder of this chapter. As for the recovery routing algorithm, the mechanism consists in creating a contour around a hole in the NoC to allow the packets to bypass it.

The X-First broadcast replication policy explained above, determines how to replicate, and forward a broadcast packet based on the input port. The recovery broadcast replication policy uses the input port, and the position of the router in the contour of the hole. This is the value in the recovery reconfiguration register, and therefore this mechanism does not need any additional configuration information. When the recovery configuration register contains NORMAL, the routers use the default X-first broadcast replication policy. Otherwise, the routers in the contour of a hole use the recovery broadcast replication policy.

The recovery firmware is in charge of modifying the reconfiguration register of the NoC routers. Algorithm 5.1 details the recovery broadcast replication policy implemented in the routers, and Figure 5.7 illustrates this replication policy by showing all the possible directions to which a router can forward a broadcast packet.

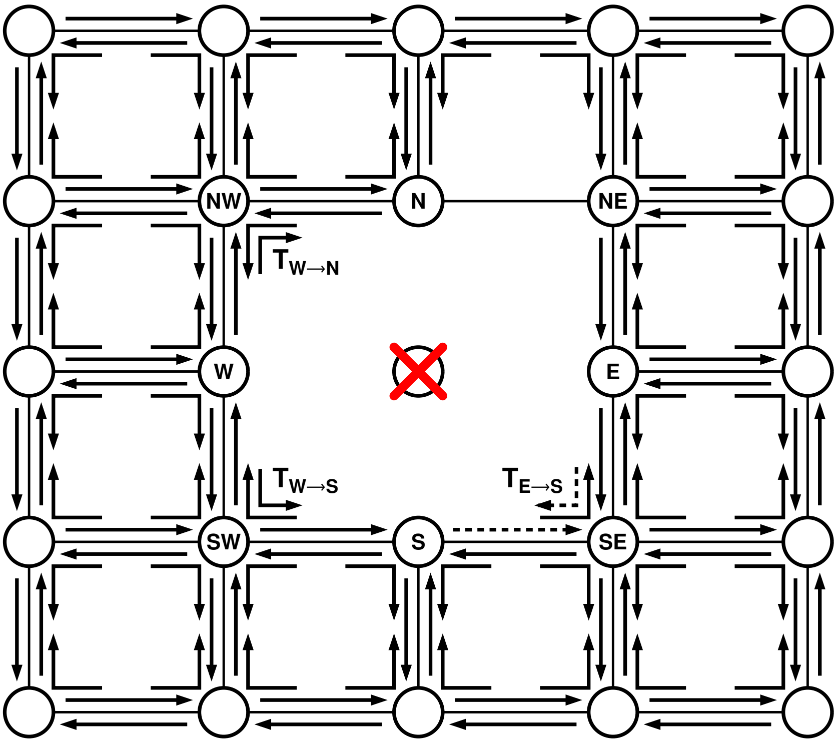


Figure 5.7 – NoC Forwarding Possibilities of the Recovery Broadcast Replication Policy

The forwarding possibilities in Figure 5.7 are represented with arrows. With respect to the X-first broadcast replication policy, the recovery broadcast replication policy adds some turns ($T_{W\to N}$, $T_{W\to S}$, and $T_{E\to S}$), it inhibits all the paths between the

N, and the NE routers, and it inhibits, in some cases, the forwarding of broadcast packets between the S and the SE routers (dashed lines in Figure 5.7).

The turns $\mathsf{T}_{\mathsf{W}\to\mathsf{N}}$, $\mathsf{T}_{\mathsf{W}\to\mathsf{S}}$, and $\mathsf{T}_{\mathsf{E}\to\mathsf{S}}$ allow a broadcast packet to bypass the hole, and reach all the clusters in the mesh. Additionally, the inhibited directions for forwarding are needed to guarantee that a packet is delivered strictly once to all clusters. This is explained further in the following.

The reason why it is necessary to inhibit the broadcast packet forwarding between the N and NE routers is illustrated in the example at the left of Figure 5.8: if the N router forwards a broadcast packet to the NE router, the clusters with $X \geq X_{NE}$ would receive the broadcast packet twice, and Property 5.4.1 would not be met. However, as it can be seen with the non-dashed lines, with the inhibition of the broadcast packet forwarding between the N and NE routers, the broadcast packet reaches once all clusters, and Property 5.4.1 is met.

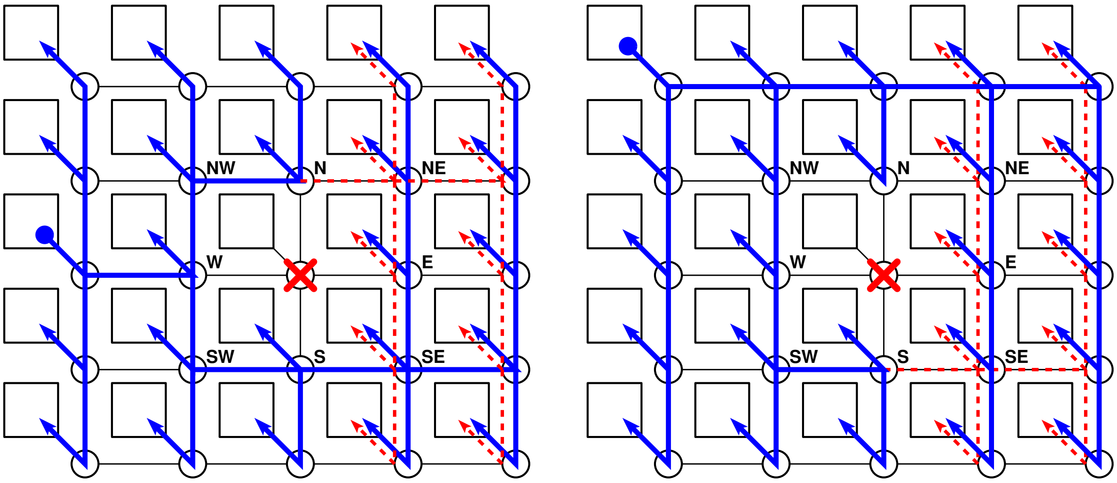


Figure 5.8 – Examples of the Recovery Broadcast Replication: It is Illustrated Why the Inhibition of the Packet Forwarding Between the N and NE Routers, and the S and SE Routers is Necessary

In the previous examples, the broadcast packet forwarding between the S and the SE routers was permitted. The example at the right of Figure 5.8 shows a case where the S → SE forwarding, and the turn $\mathsf{T}_{\mathsf{E}\to\mathsf{S}}$, need to be inhibited to guarantee the unicity of the broadcast delivery.

Therefore, the S and SE routers need a mechanism to decide whether the broadcast packet should be forwarded to S (or SE), or not. The only cases where the S and SE routers must inhibit the broadcast packet's forwarding between each other (called special forwarding), is when the broadcast source cluster is above the contour of the hole ($Y > Y_{hole} + 1$), as for the example at the right in Figure 5.8. In that case, the broadcast packet reaches all the clusters in the same row than the source, and then goes downwards and upwards through all the columns. The clusters in the south direction of the hole are reached with the turn $\mathsf{T}_{\mathsf{W}\to\mathsf{S}}$ (Figure 5.7). However, as the packet is going downwards on all columns, the routers S and SE do not need to forward the packet to each other column.

Algorithm 5.1: Recovery Broadcast Replication Policy

```
RecoveryBroadcastReplicationPolicy()
Input: $PORT_{in}$: the input port from which the broadcast packet entered the router.
Input: $CONFIG$: the reconfiguration register's Recovery Config value.
Input: $(X_L, Y_L)$: the router's local coordinates.
Input: $Special$: special bit in the broadcast packet's header.
begin
    if $PORT_{in} = \text{LOCAL}$ then
        Forward(NORTH)
        Forward(SOUTH)
        if $CONFIG \neq \text{N\_OF\_X}$ or $X_L = 0$ or $Y_L = 1$ then
            Forward(EAST)
        if $CONFIG \neq \text{NE\_OF\_X}$ or $X_L = 1$ or $Y_L = 1$ then
            Forward(WEST)
    else if $PORT_{in} = \text{NORTH}$ then
        Forward(SOUTH)
        if $CONFIG = \text{SW\_OF\_X}$ then
            Forward(EAST)
        if $CONFIG = \text{SE\_OF\_X}$ and ($X_L = 1$ or $\overline{Special}$) then
            Forward(WEST)
        Forward(LOCAL)
    else if $PORT_{in} = \text{SOUTH}$ then
        Forward(NORTH)
        if $CONFIG = \text{NW\_OF\_X}$ then
            Forward(EAST)
        if $CONFIG = \text{NE\_OF\_X}$ and ($X_L = 1$ or $Y_L = 1$) then
            Forward(WEST)
        Forward(LOCAL)
    else if $PORT_{in} = \text{EAST}$ then
        if $CONFIG \neq \text{NE\_OF\_X}$ or $X_L = 1$ or $Y_L = 1$ then
            Forward(WEST)
        Forward(NORTH)
        Forward(SOUTH)
        Forward(LOCAL)
    else if $PORT_{in} = \text{WEST}$ then
        if $CONFIG \neq \text{N\_OF\_X}$ or $Y_L = 1$) and ($CONFIG \neq \text{S\_OF\_X}$ or $\overline{Special}$)
        then
            Forward(EAST)
        Forward(NORTH)
        Forward(SOUTH)
        Forward(LOCAL)
    end
end
```

We propose to use a bit in the header flit of a broadcast packet to signal that the special forwarding must be used. As this special forwarding is needed when the broadcast packet's source is above the contour of a hole, the routers N and NE are in charge of setting the special bit when they receive a broadcast from their NORTH input port. Then, as the packet is replicated and forwarded downwards on their respective columns, the routers S and SE will receive the packets with the special bit set.

In TSAR, there were several unused bits in the header flit of a broadcast packet, therefore one of these bits is used for the special bit.

5.4.2 Verification of the Recovery Broadcast Replication Policy

The verification of the recovery broadcast replication policy is two-fold: first, we need to verify that the broadcast packets' forwarding is deadlock free. Second, we need to verify that a broadcast packet, independently of its source and the hole position, reaches all the other functional clusters of the architecture strictly once (Property 5.4.1).

In order to probe that the recovery broadcast replication policy is deadlock free, we used the theorem from Dally and Seitz [6], which states that a network is deadlock free if and only if the channel dependency graph (CDG) does not contain any cycle. The CDG is a directed graph, where the vertices are the inter-router channels, and the edges represent a possible path between two channels defined by the routing algorithm. Figure 5.9 illustrates the CDG for a router implementing the X-first routing algorithm. The dependencies of the LOCAL channels are not shown for the sake of clarity.

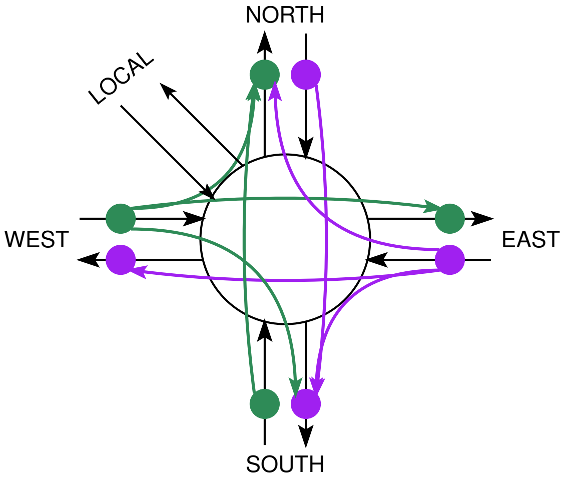


Figure 5.9 – Channel Dependency Graph of a Router Implementing the X-First Routing

As explained below, we need to support the recovery broadcast replication policy in the M2P NoC. This NoC implements both unicast and broadcast transactions, thus the CDG include the channel dependencies for both styles of communication.

We realized a C++ model with the Boost Graph Library [29] of the CDG in a 10 × 10 mesh with a hole in all the 100 possible coordinates. In none of the 100 resulting CDGs there was a cycle, so we can conclude that the recovery broadcast replication policy is deadlock free.

Regarding the verification of Property 5.4.1, we simulated a virtual prototype of a 10 × 10 mesh. Each node of the mesh consists of a broadcast initiator, a simple target, and a reconfigurable router implementing the recovery broadcast replication policy. We simulated all possible single-faulty-router scenarios: 100 possible coordinates for the hole, and 99 possible sources for the broadcast. In all 9900 simulations, the broadcast was delivered strictly once to all the clusters in the mesh.

5.5 Conclusion

In this chapter we presented the NoC reconfiguration phase of our fault-tolerance strategy.

When there are holes in the NoC, the communication between clusters is broken, and a reconfiguration mechanism is needed to modify the global routing function, and repair the intercluster communication.

We proposed a fully software-based solution to locate the holes in the NoC, and to reconfigure reliably the routers in the contour of holes. This allows modifying the global routing function, and repair the intercluster communication. We suppose that the routers implement the reconfigurable cycle-free contour routing algorithm proposed by Zhang, Greiner, and Taktak [15].

The holes' location is known from the global map of communication resources, built in the previous phase of the distributed recovery firmware (Chapter 4), and the NoC reconfiguration is performed by means of software reconfiguration messages sent by the global leader through the FFST. This fully software-based approach reduces the hardware overhead of our fault-tolerance strategy.

Additionally, in this chapter we proposed a new mechanism to reallocate the memory segment of a deactivated cluster to one of its neighbors, and therefore reduce the memory loss incurred by hard faults. This segment reallocation is implemented by modifying the global routing function of the NoC in order to reroute packets whose destination is a deactivated cluster. The mechanism benefits of the communication resources redundancy provided by the L2-L3 interconnect in the TSAR architecture.

And finally, we proposed a mechanism allowing the NoC to support broadcast communications in the presence of holes. This is an important feature because many-core architectures use generally broadcast communications for hardware cache coherency. The proposed mechanism is scalable because it is logic-based (it does not use any kind of forwarding table), and uses the same information as for the recovery routing algorithm: the relative position of the local router with respect to the hole. We called this mechanism as recovery broadcast replication policy.

The recovery broadcast replication policy has been verified to be deadlock free in any single-faulty-router topology. We also verified that the recovery broadcast replication policy guarantees that a broadcast packet reaches all the functional clusters in the architecture (independently of its source, or the position of the hole), and we verified the unicity of the broadcast delivery (each clusters receives strictly once the same broadcast packet).

Chapter 6

Experimental Results and Evaluation

Contents

6.1 Introduction

In this chapter we evaluate our proposed fault-tolerance mechanism for shared-memory many-core architectures. Our evaluation is two-folded: on the one hand, we evaluate its performance, and on the other hand, we evaluate its hardware cost.

Regarding the performance, in Section 6.4.1, we evaluate the latency of the fault-tolerance mechanism. As this mechanism is executed at every processor power-on, it is important that the latency is kept low. We also analyze the percentage of faulty cores supported by the fault-tolerance mechanism.

Regarding the hardware cost, in Section 6.5, we evaluate the hardware overhead introduced by our fault-tolerance mechanism. We pay special attention to the cost induced by the distributed ROMs containing the fault-recovery firmware.

This evaluation has been performed on a cycle-accurate SystemC [30]–[32] virtual prototype of the TSAR many-core processor. The TSAR many-core processor has a complete register-transfer level (RTL) specification, but we chose to perform the evaluation on the SystemC virtual prototype for the faster simulation speed.

Additionally, we performed a validation of the fault-tolerance mechanism by executing the Linux kernel [33] on the virtual prototype with a partially defective platform. In these experimentations, the recovery firmware locates the faulty devices, then loads the Linux kernel, and passes to this last a Device Tree Blob (DTB) structure with the modified hardware platform (all faulty devices are removed from the original description).

The outline of this chapter is the following: Section 6.2 describes the virtual prototype of the TSAR architecture, Section 6.3 describes the fault model, Section 6.4 evaluates the performance of the fault-tolerance mechanism, Section 6.5 evaluates the silicon cost, and finally, Section 6.6 concludes this chapter.

6.2 Virtual Simulation Prototype

The validation and evaluation of the proposed solution have been performed using a cycle-accurate and bit-accurate (CABA) SystemC virtual prototype of the TSAR many-core architecture [7]. Due to the high accuracy with which these models represent the hardware components, the obtained measures are almost the same as the ones that would be obtained on the hardware prototype.

The TSAR virtual prototype is developed with the SoCLib [34] library hosted by the LIP6 laboratory. SoCLib is mainly a library of SystemC simulation models (IPs) for the virtual prototyping of multi-processor system-on-chip (MP-SoC).

Of course, the virtual prototype of the TSAR architecture has been modified to introduce the various hardware modifications required by the fault-tolerance mechanism:

- One ROM per cluster, which contains the recovery firmware.
- One watchdog timer in each *L1* cache controller (Section 4.2.1).
- The *L1* and *L2* caches controllers' FSMs have been modified to support the coherence networks test mechanism (Section 4.3).
- The hardware gateway barrier in the local interconnects (Section 4.1.4).
- The scratchpad mode in the *L2* cache controller.
- The recovery routing algorithm, the segment reallocation mechanism, and the recovery broadcast replication policy were implemented in the DSPIN routers of the NoCs (Chapter 5).

Additionally, the virtual prototype was modified to allow the injection of faults (according to the fault model described in Section 6.3).

Figure 6.1 illustrates a TSAR cluster with all the additional hardware required for our fault-tolerance mechanism, and Figure 6.2 shows a logical view of the complete virtual prototype. The virtual prototype used for these experimentations is a simplified version of TSAR without the third level of caches, and the associated NoCs (between the L2 and L3 caches). However, it contains a NoC (XRAM interconnect) for interconnecting the L2 caches (called **MEMC** on Figure 6.2), IO bridges and external RAMs (XRAMs), and an external NoC (IO interconnect) interconnecting the IO bridges and external peripherals.

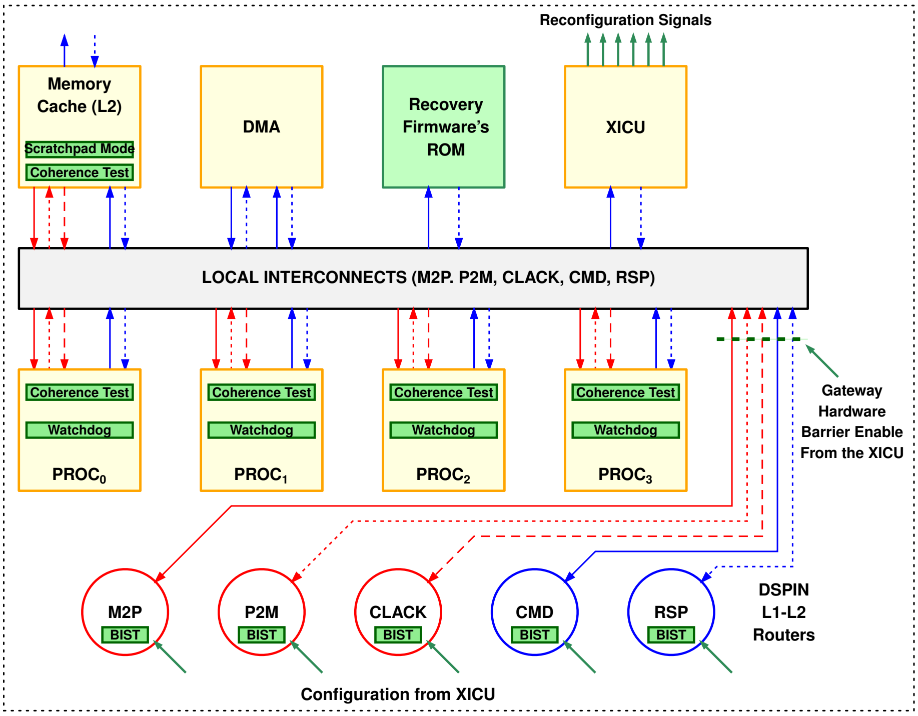


Figure 6.1 – TSAR Cluster with the Fault-Tolerance Additional Hardware

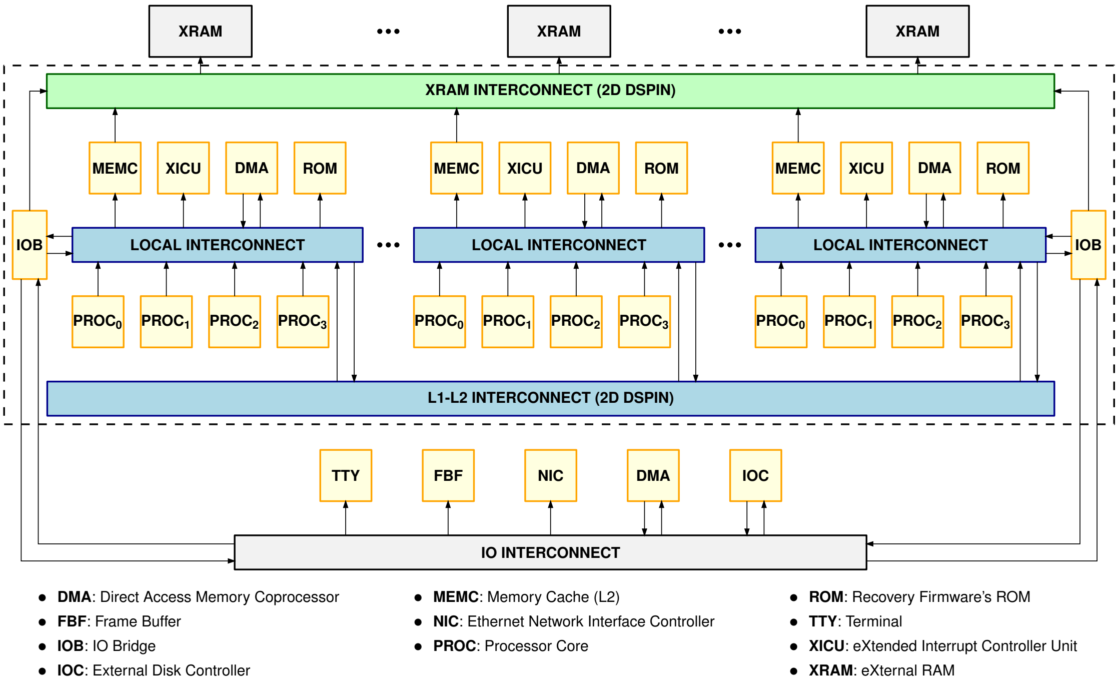

XRAM
XRAM
XRAM
XRAM INTERCONNECT (2D DSPIN)
MEMC
XICU
DMA
ROM
IOB
LOCAL INTERCONNECT
IOB
PROC$_0$
PROC$_1$
PROC$_2$
PROC$_3$
L1-L2 INTERCONNECT (2D DSPIN)
TTY
FBF
NIC
DMA
IOC
IO INTERCONNECT
• DMA: Direct Access Memory Coprocessor
• FBF: Frame Buffer
• IOB: IO Bridge
• IOC: External Disk Controller
• MEMC: Memory Cache (L2)
• NIC: Ethernet Network Interface Controller
• PROC: Processor Core
• ROM: Recovery Firmware's ROM
• TTY: Terminal
• XICU: eXtended Interrupt Controller Unit
• XRAM: eXternal RAM

Figure 6.2 – TSAR Virtual Prototype (Logical View)

6.3 Fault Model

Our simulation virtual prototype supports the injection of two kinds of faults: faulty cores or faulty routers. For each simulation, we can choose precisely which cores or routers are faulty by giving their coordinates. In the case of the NoC routers, we choose also the type (*CMD, RSP, M2P, P2M & CLACK*).

When one or more of the five routers in a cluster are faulty, the entire cluster is deactivated (all the processor cores in the cluster are idled), but the other non-faulty routers are still activated, so they can be used even if the local cluster cannot.

Because of the hardware gateway barrier mechanism implemented in each cluster (explained in Section 4.1.4), the clusters behave as black-holes at the beginning of the simulation: all incoming packets at the gateway are consumed, and no packet is produced. This hardware gateway barrier can only be released by the local cores when the cluster passes its intracluster phase. In the case of a deactivated cluster, as all its local cores are idled, the cluster behaves as a black-hole.

The actual distributed BIST of the NoC is not implemented in the virtual prototype: when a faulty router is chosen to be faulty, it is deactivated from the beginning of the simulation, and all its ports behave as black-holes.

Regarding the cores, when a faulty core is chosen to be faulty, it is self-deactivated during the execution of the Software-Based Self-Test (SBST). The implemented SBST executed during the intracluster phase is therefore very simple: in all distributed ROMs, there is an array with the IDs of the faulty cores, and each core searches in this array if there is a match for its own ID. In such case, the core self-deactivates.

6.4 Performance Evaluation

The evaluations were performed by the simulation of various sizes of the architecture: 4, 8, 16, 32, 64, 128 and 256 clusters (each cluster containing 4 processor cores).

Our entire proposed fault-tolerance mechanism consists in the following four stages:

1. Hardware-Based NoC Fault Detection.
2. Distributed Software-Based Fault Location.
3. Hardware-Assisted NoC Reconfiguration.
4. OS Loading.

The hardware-based NoC fault detection stage refers to the distributed BIST executed at the processor power-on to detect faults in the routers and channels of the NoC. As explained in Section 1.5, our fault-tolerance mechanism uses existent fault-detection solutions for both the NoC, and the processor cores. In the case of the NoC, we use as a reference the distributed BIST proposed by Zhang, Greiner, and Benabdenbi [11]. The latency of this mechanism is less than 300 clock cycles.

The distributed software-based fault location stage refers to the distributed and co-operative fault-location procedures executed by the internal cores, after the NoC BIST. These procedures allow the creation of a global map describing the operational hardware, which contains both the computational (i.e. cores and memory banks) and communication (i.e. NoC routers) devices. The latency of this stage is analyzed in Section 6.4.1.

The hardware-assisted NoC reconfiguration stage refers to the procedure executed by all the local leaders in the FFST, and coordinated by the global leader, to reconfigure the NoC, and change the global routing function. This reconfiguration allows repairing the intercluster communication, which is broken when there are holes in the NoC (faulty routers); and allows the reallocation of memory segments of deactivated clusters to its neighbors, in order to reduce the performance degradation induced by faults. The latency of this stage is analyzed in Section 6.4.2.

The latency of the OS loading stage is not analyzed, because it is strongly dependent on the throughput of the external disk controller.

Section 6.4.3 analyzes the number of tolerated faulty cores. The recovery firmware needs the many-core processor to meet some conditions in order to recover from failures. When the number of faulty cores increases, some of these conditions may be violated, and the processor is not able to recover. In this section we analyze this problem.

And finally, Section 6.4.4 describes our experimentations on loading and executing the Linux kernel in a partially defective architecture.

6.4.1 Distributed Software-Based Fault Location Latency

The procedures executed during the distributed software-based fault location stage are organized in several phases (explained in Chapter 4): processor cores' SBST, intracluster neighbors' discovery, local leader's election, intercluster neighbors' discovery, FFST (Fault-Free Spanning Tree) construction, and the distributed information gathering. At the end of these procedures, the FFST software-based communication infrastructure is built, and the global leader has a global map of the operational hardware.

FFST's Construction Latency

The FFST's construction latency considers all the phases until the FFST is actually built: from the cores' SBST, to the distributed algorithm to build the FFST.

Figure 6.3 shows this latency with respect to the size of the architecture (in terms of the number of clusters), and the percentage of randomly injected faulty cores.

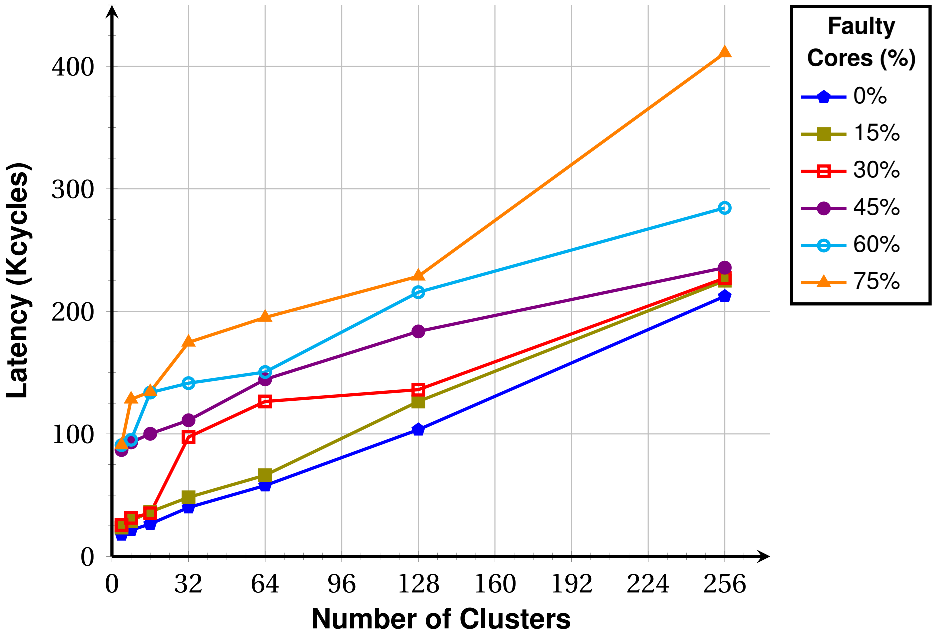


Figure 6.3 – Fault-Free Spanning Tree's Construction Latency

Two important informations can be obtained from Figure 6.3:

1. The FFST's construction latency grows linearly with the number of clusters, and it is about 225 Kcycles in a 1024-cores architecture (16 × 16 mesh of clusters).
2. There is a weak impact of the number of faulty cores on the latency. Only when the percentage of faulty cores is larger than 45% the latency is affected.

The increase of the latency when the percentage of faulty cores is larger than 45% can be explained by the increase in the number of deactivated clusters (clusters with no functional cores), that has two consequences:

- The intercluster neighbors' discovery takes longer because the functional clusters test each neighbor several times before diagnosing it as faulty.
- The latency of the FFST's construction procedure increases because it depends on the diameter of the functional network. The FFST construction is based on the propagation of messages in a neighbor-to-neighbor basis, and if the diameter of the network increases, it takes longer to reach all the functional clusters.

In a graph, the diameter is defined as the upper bound of the shortest distance between any two nodes. In a complete 2D mesh, the diameter is the Manhattan distance between two corner nodes. However, when the topology derived from deactivated clusters is strongly modified, the diameter can be increased.

An example of a strongly modified topology is shown in Figure 6.4. In the right figure, there is no deactivated cluster, and the largest round-trip is 10. In the left figure, the deactivated clusters modified the diameter, and the largest round-trip distance is 16.

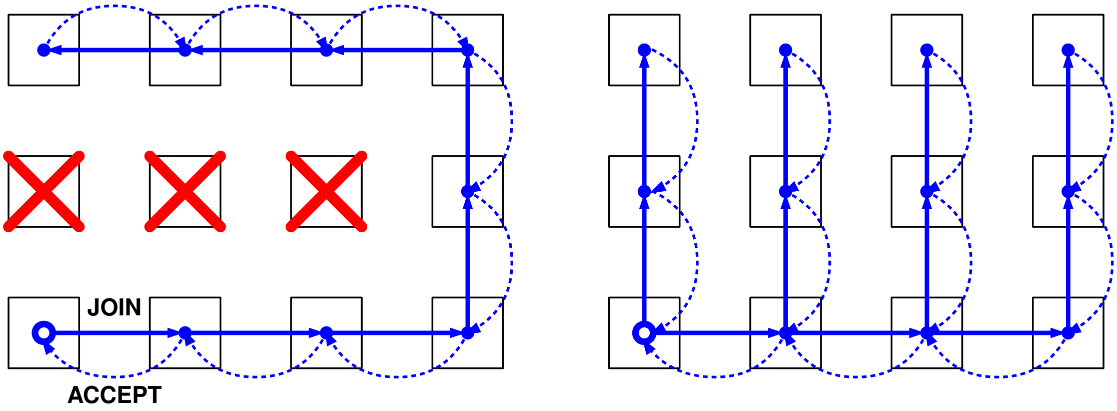


Figure 6.4 – Example of a Strongly Modified Topology Because of Deactivated Clusters

Black-Hole Location Procedure Latency

Figure 6.5 shows the latency of the black-hole location procedure. It is composed of three parts:

1. The time required to send a broadcast message through the FFST to inform all functional clusters that they must execute the black-hole location procedure.
2. The execution time of the black-hole location, which is executed in parallel by all the functional clusters in the FFST.
3. The time needed to centralize the distributed information in the global leader's cluster.

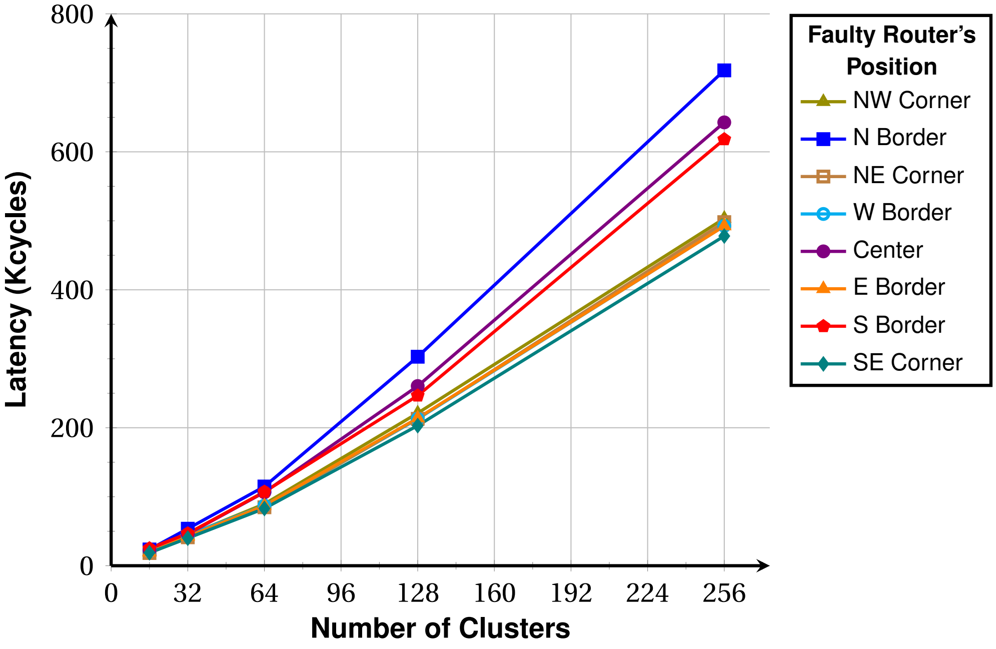


Figure 6.5 – Black-Hole Location Procedure Latency with Respect to the Position of a Faulty Router and the Number of Clusters in the Architecture

Data of Figure 6.5 are obtained from several single-faulty-router simulations. The black-hole procedure's latency is plotted with respect to the number of clusters in the architecture, and the faulty router's position. Figure 6.6 illustrates the faulty router's position used for simulations.

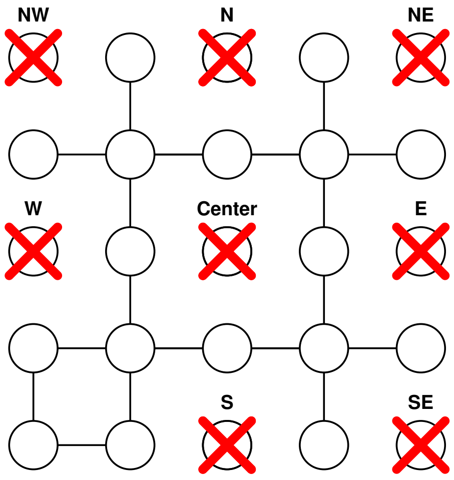


Figure 6.6 – Faulty Router's Positions for the Black-Hole Location Latency Plots

From Figure 6.5 we can conclude that the black-hole location procedure latency increases linearly with the number of clusters, and the maximum latency observed in simulations is of 720 Kcycles in a 1024-cores architecture (256 clusters). Additionally, we can observe that the latency is more affected when the faulty router is in the middle column of the mesh (N, Center, and S cases). In these cases, there is a latency increase of 35% with respect to all other cases (NW, W, NE, E and SE).

This increase can be explained because the latency of the black-hole location procedure is strongly dependent of the number of watchdog timeouts triggered within its execution. During this procedure, all the local leaders try to reach all clusters, as explained in Chapter 4, and the routers in the NoC use the X-first routing algorithm. When there is a faulty router, some X-first paths are broken, and trigger a watchdog timeout on the local leaders. The more timeouts are triggered, the more the local leaders wait before testing all the clusters. Because of the central symmetry in a 2D mesh, when the hole is in the middle column, the number of timeouts increases.

We can conclude, from Figure 6.3 and Figure 6.5, that the worst case latency of our fault-location procedure is of at most 1.2 Mcycles in a 1024-cores architecture. This value considers the latencies of both the FFST's construction and the black-hole location procedure, which allow the construction of the global map containing the computational and communication operational resources of the architecture.

6.4.2 NoC Reconfiguration Latency

In this section we analyze the latency of the NoC reconfiguration software-based procedure explained in Chapter 5. During this procedure, the global leader sends through the FFST reconfiguration commands to all the clusters in the contour of the faulty routers, in order to change the global routing algorithm of the NoC, and bypass the holes.

The NoC reconfiguration latency is plotted in Figure 6.7 with respect to the number of clusters, and the faulty router's position.

We can observe that the latency depends on both:

1. The distance between the global leader and the faulty router.
2. The number of clusters in the contour of the faulty router.

In these simulations, as the cluster (0,0) is functional, it is the global leader of the processor. Therefore, the farthest faulty router is in the NE position, and then in the N or E positions. For these positions we find the longest latencies for the NoC reconfiguration. As these are positions in the boundary of the mesh, their contours are not complete.

As we can see in Figure 6.6, the NE cluster has three clusters on its contour, and the N, and E clusters have each five clusters on their contours. However, when the faulty router is in the center of the mesh, it has a complete contour of eight clusters, and the reconfiguration latency is higher. We observed the worst NoC reconfiguration latency in this case: 329 Kcycles.

From the latency analysis of the fault-location, and NoC reconfiguration procedures, we can observe that the entire software-based fault-tolerance procedure takes at most 1.5 Mcycles in order to reconfigure the hardware, and allow the processor to recover from permanent failures.

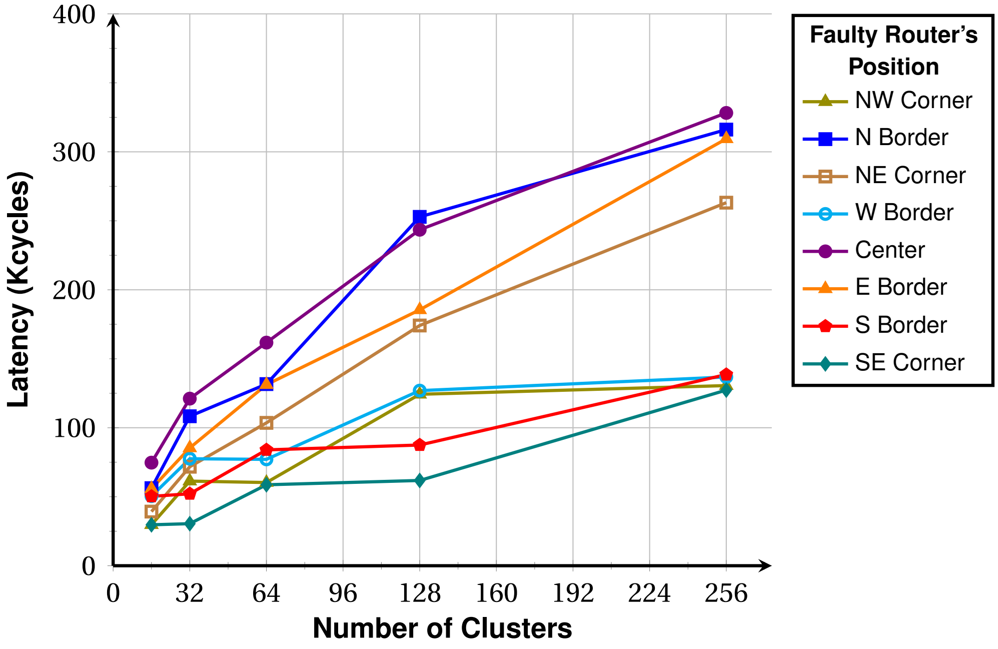


Figure 6.7 – Network-on-Chip Reconfiguration's Latency with Respect to the Position of a Faulty Router and the Number of Clusters in the Architecture

This 1.5 Mcycles value does not take into account the latencies of the NoC BIST, and the SBST. Considering the latency of the NoC BIST proposed in [11] (300 cycles), and the latency in the SBST proposed in [12] (10 Kcycles), the total latency of the fault-tolerance mechanism is still less than 1.6 Mcycles. This means that, if the clock frequency of the processor is 1 GHz, the complete fault-tolerance mechanism takes only 1.6 ms to be executed, which is negligible, compared to the operating system boot.

6.4.3 Available Computational Power

Figure 6.8 shows the available computational power with respect to the number of faulty cores in the architecture. As the clusters in the FFST represent the usable computational hardware, the available computational power is the number of cores in the FFST (reached cores). Each point in this figure is the arithmetic mean of the number of reached cores on 100 simulations with a 256-cores architecture (64 clusters). Table 6.1 shows the percentages of randomly injected faulty cores.

Ideally, the available computational power should be the complement of the faulty cores (e.g. when there are 30% of faulty cores, the available computational power should be the remaining 70% cores). This ideal case is represented with the dashed line in Figure 6.8. The solid line represents the obtained computational power.

We can observe the actual computational power follows the ideal one while the percentage of faulty cores is less than 60%. After that, all the available functional cores are not reached by the FFST, and some computational power is wasted. We can also observe that after 80% (205) of faulty cores, the computational power is zero.

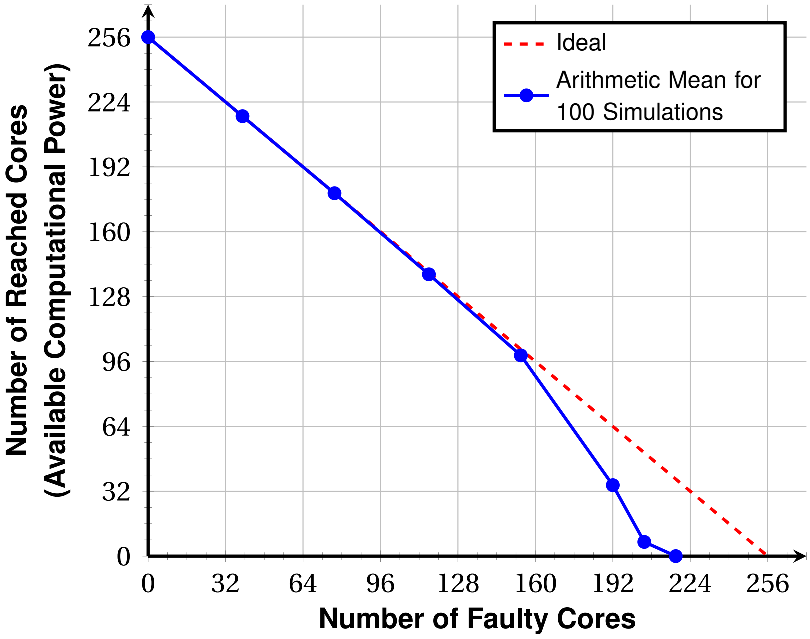


Figure 6.8 – Mean Available Computational Power on 100 Simulations with Respect to the Number of Faulty Cores in an 8 × 8 mesh (64 clusters)

Percentage (%)	Number of Faulty Cores
0	0
15	39
30	77
45	116
60	154
75	192
80	205
85	218

Table 6.1 – Number of Faulty Cores Used for the Available Computational Power Plot

In Section 4.4.2, we explained that the processor can recover from failures (have a computational power greater than 0) when there is at least one functional IO cluster, and one of these functional IO clusters can reach more than $N/2$ clusters (where N is the total number of clusters in the processor).

When the percentage of faulty cores is important, the number of deactivated clusters is also important. Therefore, there is a high probability for IO clusters to be faulty. Moreover, when the number of deactivated clusters is important, the architecture can be partitioned, and even if there are still functional IO clusters, these cannot reach more than $N/2$ clusters. In such case, the processor cannot recover from failures, and is useless.

However, we can observe that our software-based fault-tolerance solution supports up to 75% of faulty cores, which is a surprisingly good result.

6.4.4 Linux Kernel Boot in a Defective Architecture

There exists a port of the Linux kernel for the TSAR architecture [35]. This port was successfully executed in various prototypes of the architecture: a cycle-accurate SystemC virtual prototype, a FPGA-based (Field Programmable Gate Array) prototype, and a hardware emulator-based 96-cores prototype.

In the framework of this work, we have executed the Linux kernel in a partially defective virtual prototype of TSAR with 16 clusters (64 cores). Different configurations of faulty cores and faulty routers were introduced, and the Linux Kernel was successfully run in all simulations.

In these experimentations, the recovery firmware is acting as a smart distributed boot-loader executed at the processor power-on: the recovery firmware locates the faulty devices, builds a global map of the operational hardware, reconfigures the NoC when there are holes, and finally boots the Linux kernel. The booting process, which is the final stage of our recovery firmware, performs first a translation of the global map of the operational hardware to the Device Tree Blob (DTB) format, saves the resulting data structure in memory, loads the Linux kernel from the external disk controller, and passes to the kernel a pointer to the DTB. Then, the Linux kernel parses the DTB, and adapts to the subjacent hardware architecture.

6.5 Hardware Cost

In order to implement our proposed fault-tolerance mechanism for shared-memory many-core architectures, some additional hardware mechanisms need to be implemented. These mechanisms have been listed in Section 6.2.

All the mechanisms, except for the recovery firmware's ROM, have a negligible hardware cost. Regarding the ROM, we evaluate its cost with respect to the size in number of memory bits. The required capacity of the on-chip distributed ROM is 10.8 Kbytes (per cluster) without a real SBST. If we consider the SBST proposed in [12] (7 Kbytes), the total required capacity per cluster's ROM is 17.8 Kbytes.

In the TSAR architecture, the *L1* and *L2* cache controllers have a capacity of 32 Kbytes (16 Kbytes for data, and 16 Kbytes for instructions) and 256 Kbytes, respectively. As there are four *L1* cache controllers, and one *L2* cache controller per cluster, the total memory bits considering all the cache controllers in a cluster is 512 Kbytes. Therefore, the 17.8 Kbytes of the recovery firmware's ROM represents less than a 3.5% overhead on the total memory bits of a cluster.

Considering that the area of a ROM memory cell is much smaller than a SRAM memory cell (used for caches' memory), we estimate the area overhead to be less than 1% of the total silicon area.

6.6 Conclusion

In this chapter, we evaluated the fault-tolerance mechanism proposed in Chapter 4 and Chapter 5. In particular, we evaluated the latency, and the hardware cost.

Regarding the latency, we implemented the proposed mechanism in the SystemC cycle-accurate virtual prototype of the TSAR architecture. Various simulations were performed with different faulty configurations in order to estimate the worst case execution latency. We obtained that the distributed software-based fault-location procedure to build the global map of the operational hardware, and the subsequent reconfiguration of the NoC, takes at most 1.6 ms with a processor's clock frequency of 1 GHz. The latency evaluation is important because the recovery firmware is executed at each processor's power-on, but the 1.6 ms latency is negligible compared to the operating system boot latency.

Regarding the hardware cost, we estimate the hardware overhead caused by the distributed firmware to be less than 1% of the total silicon area. Therefore, we consider the hardware cost of our complete fault-tolerance mechanism to be affordable.

Another important measure that we have presented in this chapter is the percentage of tolerated faulty cores in the processor. We found that the procedure can tolerate up to 75% of faulty cores.

Finally, we have also validated our fault-tolerance mechanism by booting and executing successfully the Linux kernel in a partially defective virtual prototype of the TSAR architecture (previously reconfigured by the distributed recovery firmware).

Chapter 7

Fault-Tolerance Extension for Interconnects above the Computational Layer

Contents

The mechanism presented in chapters 4 and 5 allows the processor to support faulty cores, faulty memory banks, and faulty routers in the NoCs interconnecting L1 and L2 cache controllers. In this chapter, we propose an extension to support permanent failures in the interconnects above the computational layer.

7.1 3D NoCs Organization

As explained in Chapter 1, the TSAR architecture has two additional interconnects above the computational layer: the first is the L2-L3 interconnect, which provides the communication between L2 cache controllers, external peripherals (through the IO bridges) and L3 cache controllers, and the second is the XRAM interconnect which provides the communication between L3 cache controllers and external RAM controllers.

The L3 cache controllers are physically implemented above the computational layer using a 3D stacking technology. As one or more L3 cache layers can be implemented, the interconnects above the computational layer use thus a 3D-mesh topology. The routers in these NoCs use a modified DSPIN architecture with two additional ports: UP, and DOWN [8]. A 3D router in the L2-L3 interconnect is illustrated in Figure 7.1.

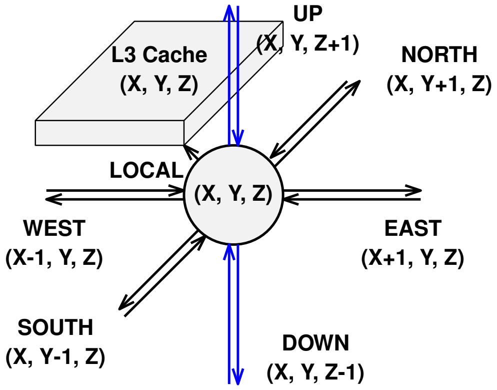


Figure 7.1 – 3D Router in the L2-L3 Interconnect

These interconnects consist each in two NoCs: one for commands, and one for responses. In the L2-L3 interconnect, the L2 cache controllers and IO bridges are the initiators, and the L3 cache controllers are the targets. In the XRAM interconnect, the L3 cache controllers are the initiators, and the external RAM controllers are the targets. During normal operation, the L2 cache controllers initiate transactions to the L3 cache controllers when there is a cache miss, or a cache line eviction; and the same for the L3 cache controllers toward the XRAM.

The computational layer is interconnected with the bottom layer of the L2-L3 NoCs. From the computational layer, the initiators are the L2 cache controllers or the IO bridges in the clusters (0, 0) and (XSIZE-1, YSIZE-1). Figure 7.2 illustrates the interconnection between the computational layer and the bottom layer of the L2-L3 NoCs. The L2 cache controllers (or IO bridges) are logically connected to the first layer of L3 caches (Z = 0), but are physically on the bottom.

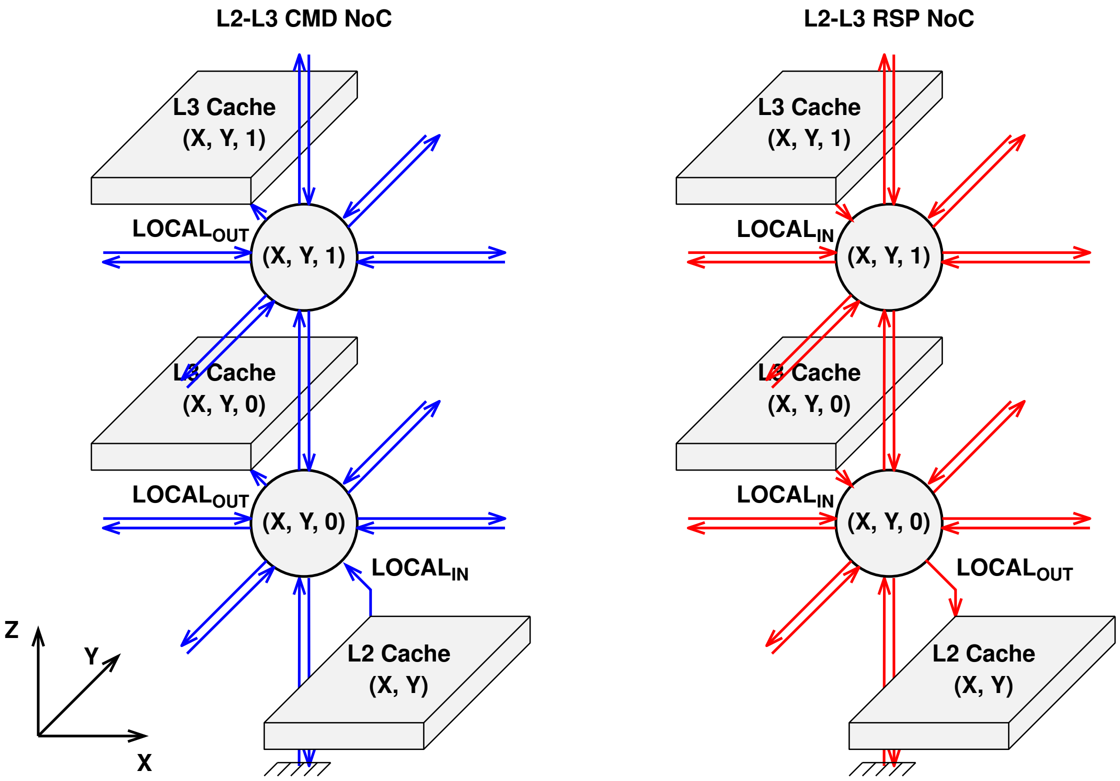


Figure 7.2 – Interconnection between the Bottom Layer of the CMD L2-L3 NoC and the Computational Layer

7.2 Physical Address Space Distribution for L3 Cache Controllers

The physical memory address space is statically distributed among the L3 cache controllers, and each manages an exclusive memory segment. This exclusive memory segment is determined by some bits of physical addresses.

Each L2 cache controller accesses only the L3 cache controllers above its cluster (L3 cache controllers with the same (X, Y) coordinates) to enforce the data access locality. Therefore, the (X, Y) coordinates of the memory segment managed by a L3 cache controller are determined from the 8 MSB (most significant bits) of the 40-bits address (as for the L2 cache controllers in the computational layer).

The third coordinate Z is determined from other bits of the address. In order to maximize the parallelism on the L3 cache controllers, we need to use the lowest bits of the address (as these are the most frequently modified). However, these bits cannot be in the bits used to index the L3 cache controllers, because the caches would be underutilized.

In TSAR, each L3 cache controller has a 1 Mbyte capacity, and implements a set-associative microarchitecture with 16-ways, 512 sets, and 128 bytes per line. Therefore, the 17 LSB of the physical address are used by the cache (bits 0–16), and we use the bits 17–18 for the Z coordinate (as shown in Figure 7.3).

Regarding the routing, the 3D DSPIN routers decode the addresses' 10 MSB to determine the target (X, Y, Z) coordinates. As the Z coordinate is not adjacent to the X,Y fields in the 40 bits address, the input network interface controllers position the Z coordinate after the X and Y coordinates in the packets' header flit, before the packets enter the NoC. Then, the output network interface controllers reposition the Z coordinate on its original bits, at the exit of the network. This Z coordinate swapping is illustrated in Figure 7.3.

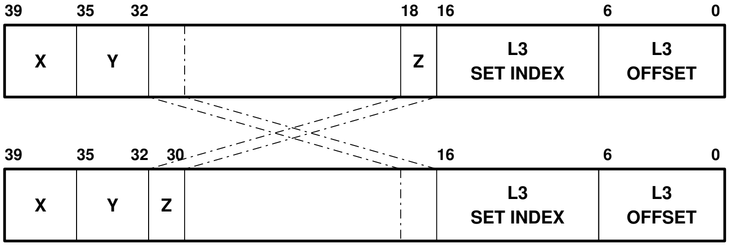


Figure 7.3 – Physical Address Format in the L2-L3 CMD NoC, and Z Coordinate Swapping

7.3 Fault-Tolerance Mechanism Overview

In order to tolerate permanent failures in these interconnects, we need to address the detection, location, containment, and reconfiguration issues.

Regarding the fault detection and containment, we propose to use the same type of hardware BIST solution, executed at the processor power-on. It deactivates the faulty routers as in the 2D mesh, and a deactivated router behaves as a black-hole.

Regarding the fault-location, we extend the recovery firmware (used in the computational layer) to locate the holes in the 3D NoCs, and build a map of the operational hardware. This extension is explained in Section 7.4.

And finally, regarding the fault-reconfiguration, in Section 7.5, we extend the reconfigurable cycle-free contour routing algorithm (detailed in Chapter 2), to bypass holes in the 3D meshes.

7.4 Software-Based Fault Location on the 3D NoCs

This section presents the software-based procedure to locate the holes in the 3D NoCs of the TSAR architecture. At the processor's power-on, the faulty routers in these NoCs are deactivated by the distributed hardware BIST, but they need to be located in order to build a global map of the operational communication resources. Then, based on this map, the NoCs can be reconfigured to change the global routing function and bypass the holes.

The idea is the same that for the black-hole location procedure in Section 4.5.2: initiators must trigger memory transactions in the L2-L3 interconnects to check which paths are broken. However, the processor cores are not directly connected to these L2-L3 NoCs.

7.4.1 Specific Test Hardware Mechanism

In order to ease the test of these NoCs, the proposed procedure uses a specific hardware mechanism in the L2 cache controllers allowing the software to trigger a test transaction in the L2-L3 interconnect. The principle was presented in Chapter 4 to test the cache-coherency NoCs in the L1-L2 interconnects.

The mechanism defines in the L2 cache controllers three memory-mapped registers: the first register allows the software to write the coordinates (X, Y, Z) of the target L3 cache controller; the second allows to store the data to send; and the third stores the status of the test transaction (command/response). The software writes the data to send, the coordinates of the target L3 cache controller, and the L2 cache controller triggers a write command on the L2-L3 command NoC. Then, the L2 cache controller waits for a response on the L2-L3 response NoC.

When the L2 cache controller triggers a test transaction, it starts a watchdog timer. If the response does not arrive before a specific threshold, the FAIL value is stored in the status register. Otherwise, if the response is received before the timeout, the L2 cache controller writes the SUCCESS value.

Then, the software reads the status register to check whether the test transaction was successful or not.

7.4.2 Black-hole Location Procedure

The black-hole location procedure for L2-L3 interconnects is executed after the one for the L1-L2 interconnects. At this point, each functional cluster is represented by a local leader (a local processor core), and the FFST is built.

Additionally, as explained in Chapter 4, the L2 cache controllers behave as scratchpads (i.e. simple memory banks) after the processor power-on. This allows the recovery firmware to be executed without using the L2-L3 interconnects that have not been tested yet.

When the local leaders receive the BLACK-HOLE LOCATION message from the global leader, they locate the holes in the L1-L2 interconnects first, and then they locate the holes in the L2-L3 interconnects. Each local leader tries to reach all the L3 cache controllers from its local L2 cache controller using the mechanism described in Section 7.4.1, and builds a Locally Discovered Routers Table (LDRT) data structure for both the L2-L3 command NoC, and the L2-L3 response NoC. These LDRTs are three-dimensional arrays, where each entry contains the status of a router in the corresponding 3D NoC.

In TSAR, the L2-L3 command NoC uses a *ZXY* routing algorithm (a.k.a. *Z-first* routing), and the L2-L3 response NoC uses a *XYThenDown* routing algorithm (a.k.a. *Z-last* routing). When a local leader reaches successfully a L3 cache controller, it tags the routers in the path in the LDRT according to these routing algorithms.

Algorithm 7.1 shows the algorithm used for tagging the routers of the L2-L3 command NoC when a test transaction is performed successfully. The algorithm to tag the routers in the L2-L3 response NoC is almost the same, but the tagging is performed according to the *XYThenDown* algorithm.

When a local leader finishes the black-hole location procedure for each NoC (L1-L2 interconnects, and L2-L3 interconnects), they centralize the gathered LDRTs in the global leader's cluster. This centralization is performed recursively from the leaves of the FFST to the root (as explained in Chapter 4).

In the LDRTs for the 3D NoCs, additionally to the status information of the routers, each local leader stores its local X and Y coordinates. This will allow the global leader to know for each L3 cache controller, which L2 cache controller can successfully reach it. This information is used during the 3D NoCs reconfiguration (explained in Section 7.7).

Algorithm 7.1: Tag ZXY Path Algorithm

```
TagZXYPath()
Input: LDRT3D: a table with the status of the routers in a 3D NoC
Input: $I_X, I_Y$: the $X$ and $Y$ coordinates of the initiator cluster
Input: $T_X, T_Y, TZ$: the $X, Y$ and $Z$ coordinates of the target L3 cache controller
begin
    /* Tag the routers on the Z direction                                  */
    /* The initiators are always the L2 cache controllers (Z = 0)          */
    for $z = 0$ to $T_Z$ do
        $LDRT3D[I_X][I_Y][z] = 1$
    end
    /* Tag the routers on the $X$ direction                                */
    $X_{min} = \min(I_X, T_X)$
    $X_{max} = \max(I_X, T_X)$
    for $x = X_{min}$ to $X_{max}$ do
        if $x \neq I_X$ then $LDRT3D[x][I_Y][T_Z] = 1$
    end
    /* Tag the routers on the $Y$ direction                                */
    $Y_{min} = \min(I_Y, T_Y)$
    $Y_{max} = \max(I_Y, T_Y)$
    for $y = Y_{min}$ to $Y_{max}$ do
        if $y \neq I_Y$ then $LDRT3D[T_X][y][T_Z] = 1$
    end
end
```

7.5 Reconfigurable Routing Algorithm for 3D NoCs

In this section we propose an extension to the reconfigurable cycle-free contour routing to support holes in a 3D NoC. The new reconfigurable routing algorithm allowing to bypass the holes is called in the remainder of this chapter as 3D recovery routing algorithm.

There are two variants for the 3D recovery routing algorithm: one for the L2-L3 CMD NoC routers, which implement the *Z-first* routing; and another for the L2-L3 RSP NoC routers, which implement the *XYThenDown* routing algorithm.

For both NoCs (CMD & RSP) the proposed solution supports one faulty router per layer. For example, if there are four L3 layers, the proposed recovery routing algorithm can support at most four faulty routers on each NoC, if each faulty router is in a different layer.

Each NoC router implements a 6-bits reconfiguration register which needs to be modified by the software to modify the routing function. Figure 7.4 shows this reconfiguration register.

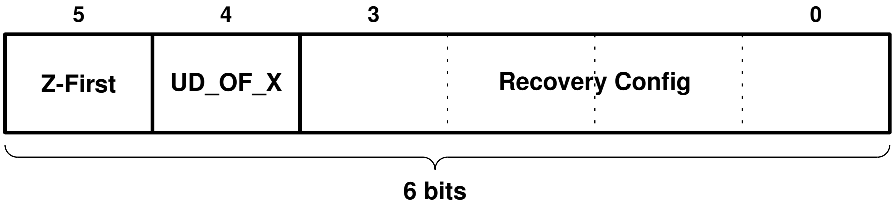


Figure 7.4 – Reconfiguration Register for L2-L3 NoC Routers

The RECOVERY_CONFIG indicates the position of the router with respect to a hole in the same layer (same XY plane), and can contain the same values as the 2D recovery algorithm: NORMAL, NW_OF_X, N_OF_X, NE_OF_X, W_OF_X, E_OF_X, SW_OF_X, S_OF_X, SE_OF_X. The UD_OF_X indicates that the router is either above (*UP*) or below (DOWN) a faulty router. And finally, the Z-FIRST bit is only used in the case of L2-L3 RSP NoC routers, and indicates that the router must use the *Z-first* routing algorithm.

7.5.1 L2-L3 CMD NoC Recovery Routing Algorithm

In the L2-L3 CMD NoC, the command packets are issued from the bottom layer. The initiators in this layer are the L2 cache controllers, or the IO bridges in IO clusters (0,0) and (XSIZE-1, YSIZE-1). As explained in Section 7.2, the L2 cache controller can only send commands to the L3 cache controllers that have the same X and Y coordinates, but the IO bridges can send commands to any L3 cache controller. By default, the routers in this NoC use the *Z-first* routing algorithm.

The *Z-first* routing algorithm consists in two phases: the first phase routes the packet to the Z coordinate of the destination (vertical phase), and the second routes the packet to the X and Y coordinates of the destination (horizontal phase).

The principle of the recovery routing algorithm for the CMD 3D NoC is the following: if the routing of a packet is in the vertical phase, and the router above the current one is faulty, the packet is forwarded to a neighbor router in the XY plane, so the packet can continue its vertical phase. And, when the packet routing is in the horizontal phase, and the current router is in the XY contour of a faulty router, the packet is forwarded according to the 2D recovery routing algorithm detailed in Chapter 2.

An example of the 3D recovery routing algorithm is shown in Figure 7.5. In these examples there are two L3 layers in a 3x3 clusters platform. In Figure 7.5 ①, the IO bridge in the cluster (0,0) addresses the L3 cache controller in coordinates (X=1, Y=2, Z=1), but there is a faulty router in coordinates (0, 0, 1). Therefore, during the vertical phase, the packet is rerouted first to the (1, 0, 0) router, so it can be forwarded up by this router. Figure 7.5 ② shows another example, where the same IO bridge addresses the same cluster (1, 2, 1), but there is a faulty router in coordinates (1, 1, 1). Therefore, during the horizontal phase, the packet is forwarded around the hole by the recovery routing algorithm.

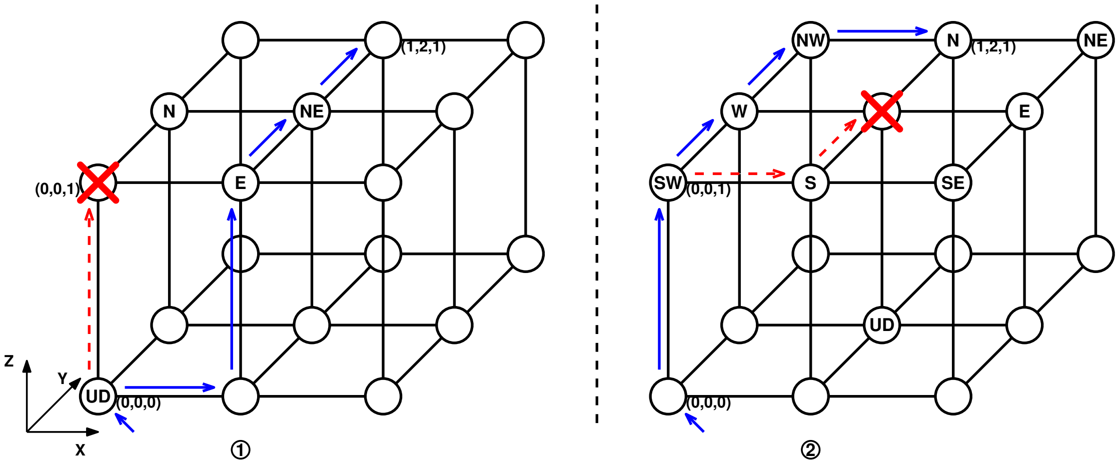


Figure 7.5 – Examples of the 3D Recovery Routing Algorithm for the L2-L3 CMD NoCs

As it can be seen in Figure 7.5, the routers that need to be reconfigured in the L1-L2 CMD NoC (when there is a hole) is the router below the hole, and the routers on the XY contour around the faulty router. In the router below, the bit UD_OF_X is set, and in the routers on the XY contour the RECOVERY_CONFIG is set with the corresponding value.

When a router in the L2-L3 CMD NoC is faulty, the corresponding L3 cache controller is deactivated. Because of the static physical distribution of the address space on the L3 cache controllers, the associated memory segment should be removed from the computational map, so the OS do not use it. However, it is possible to implement the memory segment reallocation mechanism described in Chapter 5.

7.5.2 L2-L3 RSP NoC Recovery Routing Algorithm

The 3D recovery routing algorithm for the L2-L3 RSP NoC is similar but not identical to the CMD NoC, because in the RSP NoC the response packets can be issued from any layer, and the targets are always the L2 cache controllers, or the IO bridges in the bottom layer. By default, the L2-L3 RSP NoC routers use the *XYThenDown* routing algorithm.

The *XYThenDown* routing algorithm consists of the same two phases as the *Z-first* routing algorithm, but it performs first the horizontal phase, and then the vertical phase.

When a packet is in the vertical phase but the current router is on top of a faulty router, the packet needs to be rerouted to a neighbor router in the same XY plane, so it can go down. However, as the L2-L3 NoC RSP routers use by default the *XYThenDown* algorithm, the neighbor router will return the packet and create a live-lock.

In order to solve this problem, the RSP 3D recovery routing algorithm implements the following behavior in the RSP NoC routers:

- The direct neighbors (north, south, east and west) of the router above the hole implement the *Z-first* routing algorithm.
- All the routers in the XY contour of the hole are reconfigured to implement the 2D recovery routing algorithm.
- When a router in the XY contour of a hole needs to forward a packet to the X and Y coordinates of this hole, it sends the packet to the layer below.

Figure 7.6 shows an example of the RSP 3D recovery routing algorithm. In this example, the L3 cache controllers in coordinates (1, 1, 2) and (2, 1, 2) send a response packet to the clusters (1, 1, 0) and (0, 0, 0), respectively. As the router (1, 1, 2) is above a faulty router, its bit UD_OF_X is set, and it forwards the packet to one of its direct neighbors. Then, its direct neighbors have been reconfigured to implement the *Z-first* routing algorithm, so they send the packet to the layer below. When the router in the west of the faulty router receives the packet, as the X and Y coordinates of the destination are the same as for the hole, this router sends the packet to the layer below. In the bottom layer, the packet is routed normally to its destination.

7.6 Faulty Routers in the Bottom Layer

When there is a faulty router in the bottom layer (Z = 0), the corresponding (X,Y) cluster cannot communicate with the L3 cache controllers.

When the affected cluster has been deactivated during the FFST construction procedure, its memory segment will be reallocated during the L1-L2 NoC reconfiguration. Therefore, the L3 cache controllers above the deactivated cluster are still used because they are accessed through one of its neighbor clusters.

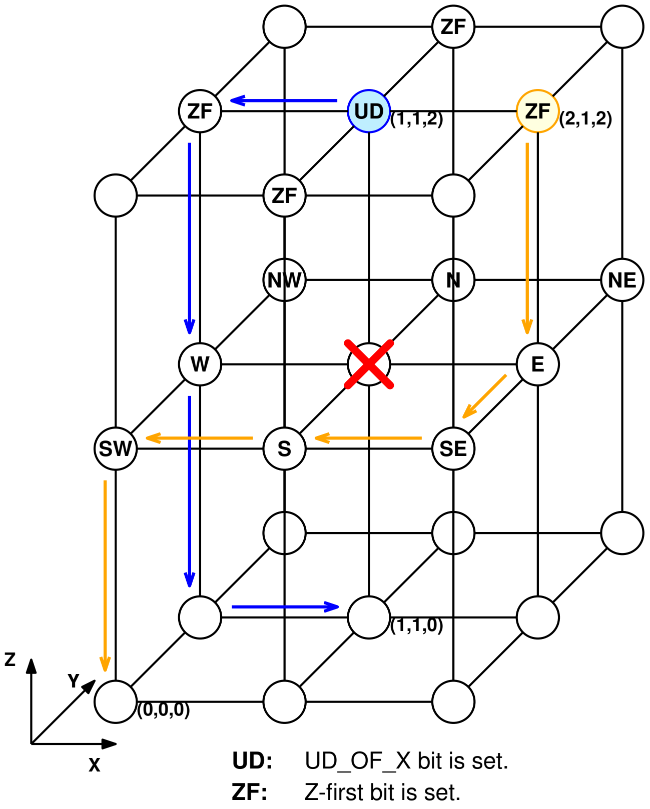


Figure 7.6 – Example of the 3D Recovery Routing Algorithm for the L2-L3 RSP NoC

However, when the affected cluster is functional, the entire memory segment associated cannot be used anymore. This is a design decision: we prefer to deactivate a memory segment instead of deactivate the four processor cores in the affected cluster.

7.7 Hardware-Assisted Reconfiguration of the 3D NoCs

This sections explains how the recovery firmware reconfigures the routers in the L2-L3 NoCs. After the black-hole location procedure (detailed in Section 7.4.2), the global leader of the FFST has the global map of operational routers in these NoCs, and can determine which routers need to be reconfigured according to the reconfigurable 3D recovery routing algorithm (detailed in Section 7.5).

The global leader is in charge of modifying the reconfiguration registers in the L2-L3 NoCs routers. As the L2-L3 routers are not directly addressable, their reconfiguration registers are implemented in the local L3 cache controller. Each L3 cache controller has thus two memory-mapped registers: one for its local L2-L3 CMD router, and another for its local L2-L3 RSP router.

In order to write in these reconfiguration registers, the global leader uses the specific hardware mechanism detailed in Section 7.4.1. When there are holes in these NoCs, some L3 cache controllers can be accessed from only a subset of the L2 cache controllers. As explained in Section 7.4.2, the LDRT data structure for the L2-L3 NoCs contains for each router which L2 cache controller can reach the corresponding L3 cache controller. Therefore, the global leader uses this information to know from which L2 cache controller it needs to reconfigure the different routers.

This reconfiguration is performed after the one of the L1-L2 NoCs, so the global leader can access directly the L2 cache controller of the functional clusters without using the FFST. This reduces importantly the reconfiguration latency of the L2-L3 NoCs.

7.8 Evaluation

In this section, we evaluate the performance, and the hardware cost of the extension to support permanent failures in the NoCs above the computational layer.

7.8.1 Performance Evaluation

The performance evaluation was made with a SystemC virtual prototype of the TSAR architecture. This virtual prototype is the same as presented in Chapter 6, but it includes additionally the L2-L3 NoCs, and the L3 cache controllers. The maximum number of clusters evaluated with this virtual prototype was 16 (4×4 mesh), and the prototype implemented four layers of L3 cache controllers.

Regarding the latency of the black-hole location procedure, the total latency to build the global map of the communication resources is increased by 20 Kcycles.

Regarding the latency of the reconfiguration procedure, as the reconfiguration of a L2-L3 NoC does not use the slow FFST software-based communication infrastructure, the reconfiguration is much faster than the reconfiguration of the L1-L2 NoCs. We observed that the reconfiguration of a L2-L3 NoC, in order to bypass a faulty router, needs at most 10 Kcycles (in contrast to the 75 Kcycles needed to reconfigure a NoC of the L1-L2 interconnect).

From these results, we conclude that the total latency of the recovery firmware is negligibly increased by the software procedures to 1) locate holes in the L2-L3 NoCs, and 2) reconfigure these NoCs when there is a faulty router. Therefore, even with the support of permanent failures in the 3D NoCs above the computational layer, the latency of the software-based mechanism is still fully-acceptable.

7.8.2 Hardware Cost

The additional hardware mechanisms needed to support permanent failures in the L2-L3 NoCs are:

1. Reconfiguration register in the 3D-DSPIN routers (6 bits).

2. Specific hardware mechanism in the L2 cache controllers to trigger test transactions in the L2-L3 NoCs (one watchdog timer, two 32-bits registers for the address and the test data, and one flip-flop for the test status).

3. Software procedures in the recovery firmware to locate the holes, and reconfigure the routers in the L2-L3 interconnect (extra 2 Kbytes).

The total firmware size passes thus from 17.8 to 19.8 Kbytes. Therefore, we consider that the complete fault-tolerance mechanism still has a very low hardware overhead.

7.9 Conclusion

In this chapter, we presented an extension to the software-based fault-tolerance mechanism in order to support in-the-field permanent failures on the NoCs above the computational layer of the TSAR architecture. Shared-memory many-core architectures implement generally various levels of interconnects to provide the communication between the processor cores, different levels of cache memories, and peripherals. Therefore, the mechanism presented in this chapter can also be used for shared-memory many-core architectures other than TSAR.

In the case of TSAR, the interconnects above the computational layer use a 3D-mesh topology. Therefore, we have extended the recovery routing algorithm presented in Chapter 2, to support holes in this kind of topologies. We have proposed a recovery routing algorithm based on the *Z-first* routing algorithm (for the L2-L3 CMD NoC), and another based on the *XYThenDown* algorithm (for the L2-L3 RSP NoC).

The evaluation of the mechanism shows that its latency and its hardware cost are negligible, so the complete fault-tolerance mechanism remains scalable.

Conclusion

In this chapter we answer the questions stated in the problem definition (Chapter 1), and then we present some future work.

How to take benefit of the many-core architectures' intrinsic redundancy in order to tolerate permanent failures in cores, memory banks, and NoC components?

The proposed fault-tolerance mechanism is based on a recovery firmware, which is executed by the internal processor cores (without any external intervention) at every processor reset. It allows the processor to support permanent failures at fabrication time (improving yield), and in the field (improving lifetime). This firmware consists in distributed and cooperative self-diagnostic procedures, which allow the cores to self-test, locate the faulty hardware components, and reconfigure the hardware.

How to locate the faulty routers, faulty cores, and faulty memory banks in order to build a map of the operational hardware devices ?

Three stages can be distinguished in the recovery firmware: distributed software-based fault location, hardware-assisted NoC reconfiguration, and OS loading.

The global map of the computational (cores and memory banks) and communication (NoC routers) operational resources is built during the distributed software-based fault location stage. This stage is subdivided into several phases, consisting each in distributed algorithms: discovery of local neighbors, election of a local leader in each cluster, discovery of direct neighbor clusters, distributed and cooperative construction of a reliable software-based communication infrastructure (called FFST), location of faulty routers in the NoCs, and centralization of the distributed information through the FFST.

We evaluated the fault-tolerance mechanism in terms of latency, hardware cost, and percentage of tolerated faulty cores. The latency and the percentage of tolerated faulty cores were investigated on the cycle-accurate virtual prototype of the TSAR many-core architecture; and the hardware cost was investigated in terms of the memory bits required by the distributed recovery firmware, and the other required fault-tolerance hardware mechanisms.

- Regarding the latency, we obtained a worst case execution latency of 1.6 Mcycles to build the global map of the operational hardware, and to reconfigure the NoC in a 1024-cores architecture. This represents 1.6 ms with a processor's clock frequency of 1 GHz. Therefore, even if the recovery firmware is executed at each processor's power-on, the 1.6 ms latency is negligible compared to the operating system boot latency. Additionally, we observed that this latency grows linearly with the number of clusters in the many-core architecture, so it is scalable.

- Regarding the hardware cost, we estimate the hardware overhead caused by the distributed firmware to be less than 1% of the total silicon area. Therefore, we consider the hardware cost of our complete fault-tolerance mechanism to be affordable.
- Regarding the percentage of tolerated faulty cores, we obtained that the fault-tolerance mechanism supports up to 75% of faulty cores.

How to reconfigure the routers reliably when the existent hardware communication infrastructure is partially defective?

The hardware-assisted NoC reconfiguration is achieved via the FFST, which is used by the elected global leader as a reliable software-based reconfiguration bus to modify the global routing function of the NoCs.

Regarding the fault-tolerant routing algorithm, the routers of the NoCs implement the reconfigurable routing algorithm defined by Zhang, Greiner, and Taktak [15], but we extended this algorithm with a mechanism to support the reallocation of the physical memory segment of a deactivated cluster to one of its neighbors, and a new routing algorithm to support broadcast communications in any single-faulty-router topology. This new broadcast routing algorithm was proved deadlock free.

Additionally, we realized an extension to the fault-tolerance mechanism to support failures in the 3D NoCs implemented above the computational layer of the TSAR architecture. This extension is two-folded: on the one hand, we defined distributed software procedures for the location of faulty routers, and their reconfiguration; and on the other hand, we propose a 3D reconfigurable routing algorithm to support faulty routers in these 3D NoCs.

How to transmit the map of the functional hardware devices to the OS so as to support its execution on a degraded architecture?

Regarding the OS loading, we use the standard Device Tree Blob (DTB) description to inform the OS of the operational hardware infrastructure at each processor reset. This DTB is built from the global map of computational resources, determined during the distributed software-based fault location stage.

This mechanism was validated by loading the Linux Kernel on a defective architecture. We used the cycle-accurate virtual prototype of TSAR including our fault-tolerance mechanism, we injected faulty components, and the distributed recovery firmware launched the Linux kernel with a dynamically built DTB, containing the description of the operational hardware infrastructure.

Future Work

Fault-Detection During the OS Execution

Our fault-tolerance mechanism locates the faulty devices, and reconfigure the architecture at each power-on. This is essential to launch the operating system on a reliable hardware platform. However, some systems can be executed during long time, and permanent failures can appear during the execution. In that case, the OS needs to implement some mechanism to automatically detect these failures, reboot the hardware, and allow the recovery firmware to reconfigure the architecture.

The OS fault-detection mechanism can reuse the procedures in the recovery firmware. This needs to be studied.

NoC Fault-Tolerance

The pre-existent NoC reconfigurable routing algorithm used in this work, and the fault-tolerant broadcast routing algorithm proposed have a low silicon cost, but support only single-faulty-router topologies.

However, in future integrated circuits with a high failure rate, it would be important to tolerate more complex irregular topologies. Some state-of-the-art fault-tolerant routing algorithms support such kind of topology, but presents a high hardware overhead. Therefore, this problem needs also to be studied further.

Appendix A

Reconfigurable Cycle-Free Routing Algorithm

Algorithm A.1 details the routing function implemented at each NoC router with the fault-tolerant routing algorithm defined by Zhang, Greiner, and Taktak [15].

This fault-tolerant routing algorithm supports any single-faulty-router topology in a 2D mesh. Each router contains a 4-bits configuration register, which modifies the routing function of the router according to its position in the contour around a hole. This configuration register can contain nine possible values: NORMAL, N_OF_X, NE_OF_X, E_OF_X, SE_OF_X, S_OF_X, SW_OF_X, W_OF_X, NW_OF_X.

When the configuration register contains NORMAL, the router is not in the contour around a hole, and implements the X-first routing function. Otherwise, the router is in the contour around a hole, and the configuration value determines the position with respect to this hole.

The reconfigurable cycle-free routing function at each router forwards the packets according to the local coordinates of the router (X_L, Y_L), the destination coordinates (X_D, Y_D), and the configuration value *CFG*.

Algorithm A.1: Reconfigurable Cycle-Free Routing Algorithm

```
RecoveryRoutingAlgorithm()
Input: CFG: the reconfiguration register's Recovery Config value.
Input: ($X_D$, $Y_D$): the destination coordinates.
Input: ($X_L$, $Y_L$): the local coordinates.
begin
    if $X_D > X_L$ then
        if $CFG \in \{ NE\_OF\_X, E\_OF\_X, SE\_OF\_X, S\_OF\_X, NORMAL \}$ then
            Forward($EAST$)
        else if $CFG = N\_OF\_X$ then
            if $(Y_L = 1)$ or $(X_L = 0)$ or $(Y_D \geq Y_L)$ or $(X_D > (X_L + 1))$ then Forward($EAST$)
            else Forward($WEST$)
        else if $CFG = NW\_OF\_X$ then
            if $(Y_L = 1)$ or $(Y_D \geq Y_L)$ or $(X_D > (X_L + 2))$ then Forward($EAST$)
            else Forward($SOUTH$)
        else if $CFG = W\_OF\_X$ then
            if $(Y_L = 0)$ or $(Y_D > Y_L)$ then Forward($NORTH$)
            else Forward($SOUTH$)
        else if $CFG = SW\_OF\_X$ then
            if $(Y_D \leq Y_L)$ or $(X_D > (X_L + 1))$ then Forward($EAST$)
            else Forward($NORTH$)
    else if $X_D < X_L$ then
        if $CFG \in \{N\_OF\_X, NW\_OF\_X, W\_OF\_X, SW\_OF\_X, S\_OF\_X, NORMAL\}$
        then
            Forward($WEST$)
        else if $CFG = NE\_OF\_X$ then
            if $(X_D < (X_L - 1))$ or $(Y_D \geq Y_L)$ then Forward($WEST$)
            else Forward($SOUTH$)
        else if $CFG = SE\_OF\_X$ then
            if $(X_L = 1)$ or $(Y_D > (Y_L + 1))$ then Forward($NORTH$)
            else Forward($WEST$)
        else if $CFG = E\_OF\_X$ then
            if $(Y_L = 0)$ or $((X_L = 1)$ and $(Y_D > Y_L))$ then Forward($NORTH$)
            else Forward($SOUTH$)
    else if $Y_D > Y_L$ then
        if $CFG \neq S\_OF\_X$ then Forward($NORTH$)
        else if $X_L \neq 0$ then Forward($WEST$)
        else Forward($EAST$)
    else if $Y_D < Y_L$ then
        if $CFG \neq N\_OF\_X$ then Forward($SOUTH$)
        else if $X_L \neq 0$ then Forward($WEST$)
        else Forward($EAST$)
    end
    Forward($LOCAL$)
end
```

Bibliography

[1] S. Borkar, “Thousand Core Chips: A Technology Perspective”, in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC ’07, New York, NY, USA: ACM, 2007, pp. 746–749, ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278667. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278667.

[2] J. Henkel, L. Bauer, N. Dutt, *et al.*, “Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends”, in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC ’13, New York, NY, USA: ACM, 2013, 99:1–99:10, ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488857. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488857.

[3] International Technology Roadmap for Semiconductors (ITRS). (2013). Process Integration, Devices, and Structures Report, [Online]. Available: http://www.itrs.net.

[4] W. J. Dally and B. Towles, “Route Packets, Not Wires: On-Chip Interconnection Networks”, in *Design Automation Conference, 2001. Proceedings*, IEEE, 2001, pp. 684–689. DOI: 10.1109/DAC.2001.156225.

[5] D. M. Chapiro, “Globally-Asynchronous Locally-Synchronous Systems (Performance, Reliability, Digital)”, AAI8506166, PhD thesis, Stanford, CA, USA, 1985.

[6] W. Dally and C. Seitz, “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”, *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547–553, May 1987, ISSN: 0018-9340. DOI: 10.1109/TC.1987.1676939.

[7] LIP6. (2015). TSAR, [Online]. Available: https://www-soc.lip6.fr/trac/tsar.

[8] E. Guthmuller, I. Miro-Panades, and A. Greiner, “Adaptive Stackable 3D Cache Architecture for Manycores”, in *2012 IEEE Computer Society Annual Symposium on VLSI - ISVLSI*, Amherst, MA, United States, Aug. 2012, pp. 39–44. DOI: 10.1109/ISVLSI.2012.36. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00743530.

[9] I. Miro Panades, A. Greiner, and A. Sheibanyrad, “A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach”, in *1st International Conference on Nano-Networks and Workshops - NanoNet ’06*, Sep. 2006, pp. 1–5. DOI: 10.1109/NANONET.2006.346219.

[10] B. Johnson, “Fault-Tolerant Microprocessor-Based Systems”, *IEEE Micro*, vol. 4, no. 6, pp. 6–21, Dec. 1984, ISSN: 0272-1732. DOI: 10.1109/MM.1984.291277.

[11] Z. Zhang, A. Greiner, and M. Benabdenbi, "Fully distributed initialization procedure for a 2D-Mesh NoC, including off-line BIST and partial deactivation of faulty components", in *2010 IEEE 16th International On-Line Testing Symposium*, IEEE, Jul. 2010, pp. 194–196, ISBN: 978-1-4244-7724-1. DOI: 10.1109/IOLTS.2010.5560209.

[12] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-Based Self-Testing of Embedded Processors", *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, 2005.

[13] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor Software-Based Self-Testing", *IEEE Design and Test of Computers*, no. 3, pp. 4–19, 2010.

[14] S. Di Carlo, P. Prinetto, and A. Savino, "Software-Based Self-Test of Set Associative Cache Memories", *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, 2011.

[15] Z. Zhang, A. Greiner, and S. Taktak, "A Reconfigurable Routing Algorithm for a Fault-Tolerant 2D-Mesh Network-on-Chip", in *Proceedings of the 45th annual conference on Design automation - DAC '08*, New York, New York, USA: ACM Press, 2008, pp. 441–446, ISBN: 9781605581156. DOI: 10.1145/1391469.1391584.

[16] L. Zhang, Y. Han, Q. Xu, and X. Li, "Defect Tolerance in Homogeneous Manycore Processors Using Core-Level Redundancy with Unified Topology", in *Design, Automation and Test in Europe, 2008. DATE '08*, ACM, Mar. 2008, pp. 891–896. DOI: 10.1109/DATE.2008.4484787.

[17] P. Zajac and J. Collet, "Production Yield and Self-Configuration in the Future Massively Defective Nanochips", in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems - DFT '07*, Sep. 2007, pp. 197–205. DOI: 10.1109/DFT.2007.34.

[18] J. H. Collet, P. Zajac, M. Psarakis, and D. Gizopoulos, "Chip Self-Organization and Fault Tolerance in Massively Defective Multicore Arrays", *IEEE Transactions On Dependable and Secure Computing*, vol. 8, no. 2, pp. 207–217, 2011.

[19] A. Kamran and Z. Navabi, "Online Periodic Test Mechanism for Homogeneous Many-Core Processors", in *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2013, pp. 256–259. DOI: 10.1109/VLSI-SoC.2013.6673285.

[20] F. Chaix, D. Avresky, *et al.*, "Fault-Tolerant Deadlock-Free Adaptive Routing for Any Set of Link and Node Failures in Multi-cores Systems", in *9th IEEE International Symposium on Network Computing and Applications (NCA)*, Jul. 2010, pp. 52–59. DOI: 10.1109/NCA.2010.14.

[21] ——, "A Fault-Tolerant Deadlock-Free Adaptive Routing for on Chip Interconnects", in *Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2011, pp. 1–4. DOI: 10.1109/DATE.2011.5763303.

[22] C. Cunningham and D. Avresky, "Fault-Tolerant Adaptive Routing for Two-Dimensional Meshes", in *First IEEE Symposium on High-Performance Computer Architecture*, 1995, pp. 122–131. DOI: 10.1109/HPCA.1995.386549.

[23] C. J. Glass and L. M. Ni, "Fault-Tolerant Wormhole Routing in Meshes Without Virtual Channels", *IEEE transactions on parallel and distributed systems*, vol. 7, no. 6, pp. 620–636, 1996.

[24] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie, "Segment-Based Routing: an Efficient Fault-Tolerant Routing Algorithm for Meshes and Tori", in *20th International Parallel and Distributed Processing Symposium, IPDPS*, Apr. 2006, 10–pp. DOI: 10.1109/IPDPS.2006.1639341.

[25] S. Rodrigo, S. Medardoni, J. Flich, D. Bertozzi, and J. Duato, "Efficient Implementation of Distributed Routing Algorithms for NoCs", *IET Computers Digital Techniques*, vol. 3, 460–475(15), 5 Sep. 2009, ISSN: 1751-8601.

[26] D. Gibson and B. Herrenschmidt, "Device Trees Everywhere", *OzLabs, IBM Linux Technology Center*, 2006.

[27] G. Likely and J. Boyer, "A Symphony of Flavours: Using the device tree to describe embedded hardware", in *Proceedings of the Linux Symposium, Volume Two*, Ottawa, Canada, 2008, pp. 27–38.

[28] A. van de Goor, "Using march tests to test SRAMs", *Design Test of Computers, IEEE*, vol. 10, no. 1, pp. 8–14, Mar. 1993, ISSN: 0740-7475. DOI: 10.1109/54.199799.

[29] J. Siek, L.-Q. Lee, and A. Lumsdaine. (2015). Boost graph library, [Online]. Available: http://www.boost.org/libs/graph/.

[30] IEEE Computer Society, *1666-2011 IEEE Standard for SystemC Language Reference Manual*, IEEE, 2012, ISBN: 978-0-7381-6801-2 STD97162.

[31] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC^{TM}*. Springer Science and Business Media, 2002.

[32] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*. Springer Science and Business Media, 2009, vol. 71.

[33] D. P. Bovet and M. Cesati, *Understanding the Linux kernel.* O'Reilly Media, Inc., 2005.

[34] LIP6. (2015). SoCLib, [Online]. Available: http://www.soclib.fr.

[35] J. Porquet, A. Greiner, and C. Fuguet T., "Porting the Linux kernel to the TSAR manycore architecture", *Design, Automation and Test in Europe University Booth (DATE)*, 2015.