An Implementation of a Distributed Fault-Tolerant Mechanism for 2D Mesh NoCs

César Marcon[1], Alexandre Amory[1], Thais Webber[2], Felipe T. Bortolon[1], Thomas Volpato[1], Jader Munareto[1]

[1]Faculty of Informatics / [2]Faculty of Electrical Engineering
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Av. Ipiranga 6681, Porto Alegre, Brazil

cesar.marcon@pucrs.br

Abstract—**Advances in design integration have enabled the integration of large Multiprocessor Systems-on-Chip (MPSoC). Such systems are prone to the execution of complex applications if high degree of parallelism is employed on the communication infrastructure. Network-on-Chip (NoC) has emerged as a new communication paradigm for large MPSoCs with advantages such as the increase of reliability on components interactions. However, device's integration may convey few shortcomings during MPSoC manufacturing and operation, for instance, the vulnerability to faults. This paper describes Phoenix, which is a direct mesh NoC with fault detection scheme. The proposed architecture explores a fault-tolerant mechanism, which is implemented in a distributed manner as a fault monitor on processors and routers. Results demonstrate that Phoenix can be scalable in view of the stabilization time regarding to faults incidence, allowing MPSoC operation even with the occurrence of a large number of faults.**

Keywords - Fault tolerance, MPSoC, NoC.

I. INTRODUCTION

Deep submicron technologies have enabled the integration of billions of transistors for the construction of complete systems over on a single chip, called Systems-on-Chip (SoCs). With the advantages of these technologies, some drawbacks are found such as the occurrence of faults originated in the complex manufacturing process, or after manufacture during system execution.

SoCs can be often implemented with several Processing Elements (PEs) operating in parallel to cope with application requirements and guarantee high data throughput. These SoCs are usually called Multiprocessor SoC (MPSoC), whose architecture also requires an efficient communication approach such as Network-on-Chip (NoC) that are typically designed to meet performance requirements [1]. The parallelism of NoC communications allows redundant communication between resources. In case of path failure, alternative routes can take place; though the consequence is a reduction on the parallelism with a probable latency increase despite application functionalities may appear normal for end-user.

One challenge in applying fault tolerance on MPSoCs is the research of fault detection mechanisms for improvement in recovering, and sometimes, prevention from these faults. During MPSoC operation, several faults can occur and the application of a fault-tolerant NoC is indicated to reduce the probability of application stall. The efficient design of monitoring mechanisms may determine whether the system is able to withstand several detected faults as well as the delay imposed when recovering the system.

This paper describes the design of Phoenix, a fault-tolerant 2D mesh NoC that implements mechanisms for detecting faults and disseminate this information to all PEs. Phoenix presents source routing tables to fulfill this objective, supporting detection of manufacturing faults and faults occurred during system execution. Following sections address (i) the NoC architecture, (ii) the mechanism for detecting faults through link analysis during idle periods, and (iii) the distributed algorithm to report faults. The routing algorithm is not addressed in this work.

The paper is organized as follows. Section II presents some related works. Section III describes the Phoenix architecture. Section IV presents the fault-tolerant mechanisms implemented on Phoenix. Section V discusses experimental results in terms of stabilization times for faults report on varied NoC sizes as well as the implementation of Phoenix mechanisms. Finally, in Section VI we conclude our contribution.

II. RELATED WORKS

Fault-tolerant methods for NoCs can be classified in two categories in terms of redundancy: (*i*) the methods based on extra redundancy to the NoC, which include spare wires, spare routers and backup NoC paths [2]; (*ii*) the methods where extra logic is not used to increase the NoC redundancy, but to exploit the natural path redundancy existing in most network topologies. For instance, for a single pair of communicating points, a mesh network typically have multiple possible paths (excluding path restrictions caused by a given routing algorithm). However, a single fault is sufficient to crash an entire system without proper methods for NoC fault detection, diagnose, and recover. Thus, multiple paths are still not sufficient to build resilient systems. In addition, NoCs are commonly used in MPSoCs due to their superior performance in terms of bandwidth and scalability. The use of NoC-based MPSoCs for fault-tolerant applications provides a large research opportunity since the fault-tolerant methods can be implemented in hardware, software, or a combination of both. Next, we present related works that are summarized and compared on Table I and on Table II.

Vicis [3] uses the inherent redundancy of most networks to keep the system functionality with lower hardware overhead compared to approaches based on triple modular redundancy

978-1-4799-2409-7/13$31.00 ©2013 IEEE

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.

(TMR). Each router has Built-In Self-Test to diagnose faults and to reconfigure the hardware bypassing defective regions. The method is entirely implemented in hardware, presenting low area overhead with greater fault tolerance than TMR methods.

TABLE I - RELATED WORK SUMMARY (PART I).

Work	Fault location	Implementation	Base approach
[3]	router	hardware	bypass, BIST, ECC
[4]	link	hardware	routing table
[5]	router	hardware	turn-based routing algorithm
[6]	router	both	path search, 2 NoCs
[7]	router	hardware	routing table
This	link	both	routing table

TABLE II - RELATED WORK SUMMARY (PART II).

Work	Fault duration	Means to dependability	Quantity of faults
[3]	permanent	diagnose, reconfiguration	50% of routers
[4]	permanent	reconfiguration	10% of the links
[5]	permanent	reconfiguration	1 faulty router
[6]	permanent, partially transient	reconfiguration	large
[7]	permanent	reconfiguration	-
This	permanent, partially transient	detection, reconfiguration	large

Fick et al. [4] present a routing algorithm, which configures the network to avoid the fault components maintaining the correct functionality. Most fault-tolerant routing algorithms circumvent the faulty region with restrictions that might cause healthy router to be disabled, i.e. reducing the network performance. The proposed method, based on routing tables and no virtual channel can support 10% of faulty links.

Zhang et al. [5] propose a method to avoid deadlock, which uses two networks and each node is connected to two routers. When a single link is declared faulty, the adjacent routers are entirely disabled. The authors adopted a turn-based fault-tolerant approach to avoid cycles. They proved that the approach is deadlock free for any one-faulty-router.

Wachter et al. [6] propose the use of a second dedicated network to find a fault-free path between two nodes. The routers have a configuration register that can switch on/off each router port in case of faults. The faulty ports, which are turned-off, are not able to propagate the search for path. This way, just a fault-free path is able to propagate (like a broadcast) the searches for fault-free paths. This approach enables to find any path regardless the number of faulty ports, as long as there is at least one healthy path. The broadcast propagation style ensures that it can be used in any network topology.

Feng et al. [7] proposed a routing algorithm to reconfigure the routing table in the presence of faults. An optimized hierarchical approach is also proposed, reducing the number of table entries. Still, the number of required data in the routing table is large. Thus, this approach is viable only for small to medium networks.

III. PHOENIX ARCHITECTURE

Figure 1 illustrates the fault-tolerant mesh NoC architecture of Phoenix implemented in hardware and software, employing routing tables for source routing decisions and fault-tolerant distributed mechanisms. *OsPhoenix* is the software part, which is a communication device placed inside each PE's operating system. *OsPhoenix* performs routing algorithms to fill the routing table according to the PE position and the faulty links.

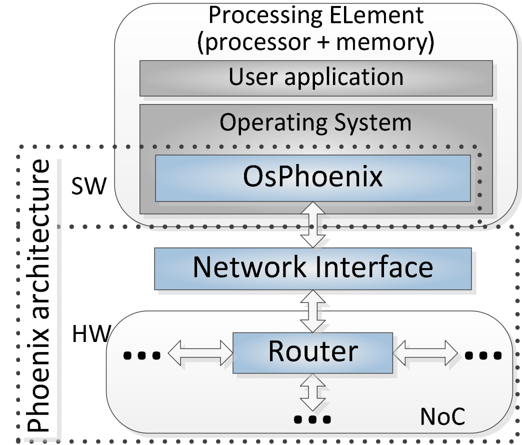

Figure 1 - The Phoenix router architecture.

A. NoC Topology

The routers and PEs connections implemented by bidirectional links define NoC physical topology. Phoenix is a direct 2D mesh NoC topology, consisting of $m \times n$ routers interconnecting PEs placed along with them.

B. Router Interface

Figure 2 shows a bidirectional link between two routers. The output ports are composed of the following signals: (i) *clockTx* that synchronizes data transmission; (ii) *tx* that controls the data availability; (iii) *dataOut*, which is a bus containing data to be sent; and (iv) *creditIn*, which is a control signal that indicates the buffer availability. In addition, the input ports are composed of the following signals: (i) *clockRx*; (ii) *rx*; (iii) *dataIn*; and (iv) *creditOut*, which are the counterpart of the output port signals, respectively. Therefore, each bidirectional link has 6 control signals and 2×flit[1] data signals.

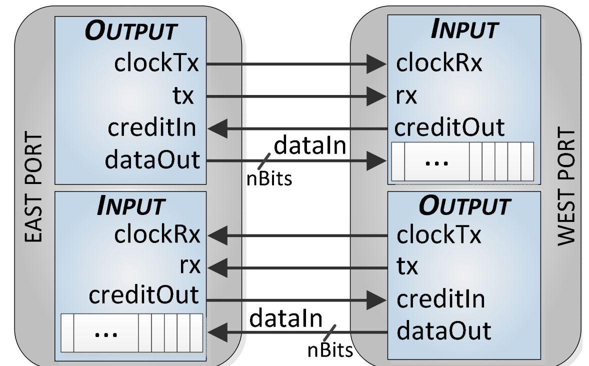

Figure 2 – Example of bidirectional link between routers.

[1] *The flit size of Phoenix is equal to the phit size.*

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.

C. Router Switching and Flow Control

Phoenix NoC implements wormhole switching method, which implies dividing packets into flits needing small buffers for data storing. The flit size of Phoenix NoC is customizable, and the number of flits in a packet is limited to $2^{(\text{flit size in bits})}$. Additionally, Phoenix NoC employs credit-based flow control. In this mechanism, if there is an available space in the receiver input buffer, the receiver router informs the transmitter router through *creditOut*/*creditIn* (Figure 2) signal, and the transmitter interprets as an available credit enabling a flit transmission in a single clock cycle.

D. Router Architecture

Figure 3 shows the Phoenix router architecture that is logically composed of three modules: (i) communication; (ii) routing; and (iii) fault management.

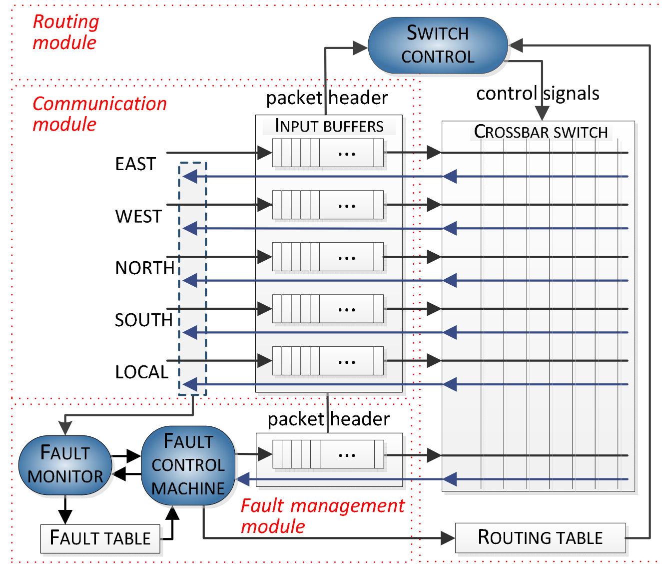

Figure 3 - The Phoenix router architecture.

Communication module encompasses four bidirectional ports (i.e. NORTH, SOUTH, EAST and WEST) dedicated to interconnect routers, and a bidirectional port (i.e. LOCAL) that enables the communication between the router and its local PE.

Each bidirectional port has an input and an output link, and the input communication is buffered with a circular FIFO with configurable depth for temporary data storage, which is used when the routing path is congested by other packets.

Routing module is controlled by *Switch control* circuit that performs the packets routing and arbitration according to the *packet header* and the *Routing table*. The arbitration is a dynamic rotating policy implemented with Round Robin algorithm to ensure that all incoming requests are processed, i.e. preventing starvation phenomenon.

This algorithm takes in average three clock cycles to address a routing request dispatched by the reading of the *packet header*. If the routing algorithm enables the communication, the *Switch control* commands the *Crossbar switch*, through the *control signal*, to establish the connection between input port and the desired output port.

Fault management module includes the *Fault control machine* that is the main circuit for fault-tolerant operation. It searches for control packets in all input ports, and takes decisions according to the command code (refer to Section III.F). For instance, *Fault control machine* may receive a command from *OsPhoenix* to fill the *Routing table*. Additionally, the *Fault monitor* circuit is responsible for detecting defective output link, and set the links status on the *Fault table*, which is a 4-field vector. Each field is used to store the operation status of NORTH, SOUTH, EAST and WEST output links, where each field contains two bits to inform if the link is (i) not verified, (ii) faulty or (iii) operating properly.

E. Routing Algorithm

Phoenix is a source routing NoC whose path is computed according to the *Routing table* content, which starts filled with XY routing. However, depending on the faults occurrence, the *OsPhoenix* searches for new routing paths that are deadlock free, which modifies the *Routing table* content. *OsPhoenix* routing algorithm is based on a Region Based Routing [8], which is a technique that group target addresses into regions aiming to reduce the routing table size. With a minimum of four regions, the *Routing table* may select several paths even in the presence of faults. Nevertheless, increasing *Routing table* size, Phoenix may provide more optimized paths searching for a possible minimum path. This algorithm is not described here, since it is not focus of this work.

F. Packet Format

Each field of a Phoenix packet is exactly 1-flit length, and despite this length is user defined, the NoC requires a minimum flit of 8-bits length to support in the same flit the control flag and addressing of 64 PEs. Figure 4 shows that Phoenix employs two types of packets: (i) *data packet*, which carries the PEs messages; and (ii) *control packet*, which is used by the router control mechanisms.

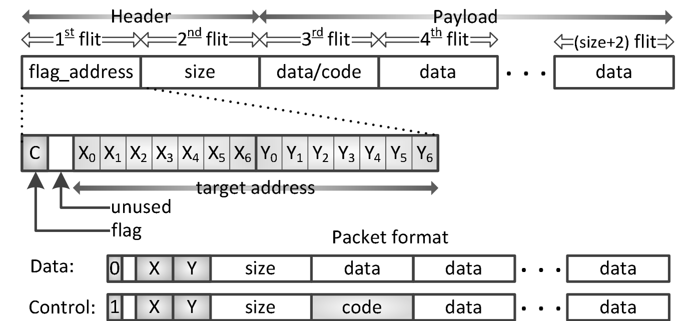

Figure 4 – Formats of Phoenix packets.

Both packet types contain a header that encloses two flits: (i) *flag_address*, which is the first flit of the header composed of (a) 1-bit flag to define the packet type (i.e. 0 is a *data packet* and 1 is a *control packet*) and (b) the XY target address (e.g. , Figure 4 exemplifies the target address distributed on a flit of 16-bits length); and (ii) *size*, which is the second flit of the header containing the quantity of flits that composes the packet payload. Whereas the *data packet* payload is completely transparent to NoC operation, the first flit of *control packet* payload is a command code used to control the routers status/operation.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.

The *code field* may carry the following commands detailed next in Section IV:

- RD_FAULT_TAB - *OsPhoenix* uses it to read the *Fault table*. When *Fault control machine* receives this command, it replies with the FAULT_TAB command containing the *Fault table* in the second flit of the payload;
- WR_FAULT_TAB - *OsPhoenix* sets the links fault status on the *Fault table*;
- TEST_LINKS –*OsPhoenix* tells the *Fault monitor* to test all links and set the *Fault table*;
- RST_FAULT_TAB - *OsPhoenix* resets the *Fault table* (i.e. all links are marked as without faults). This code is transmitted to all *OsPhoenix* running on neighbor PEs;
- RST_ALL_FAULTS - this code is transmitted to all *OsPhoenix* running on neighbor PEs performing a distributed way to reevaluate the status fault of all NoC links;
- TR_FAULT_TAB - *OsPhoenix* transmits the *Fault table*, enclosed into the payload, to another *OsPhoenix* running on a neighbor PE;
- RD_ROUT_TAB - *OsPhoenix* reads the *Routing table* (these code is normally applied during NoC debugging);
- ROUT_TAB - the router replies the RD_ROUT_TAB code inserting the *Routing table* in payload flits. Notice that the *Routing table* size depends on the number of regions defined by the routing mechanism;
- WR_ROUT_TAB - *OsPhoenix* sets the *Routing table* with the values produced by the routing algorithm. The *Routing table* is enclosed on the packet payload;
- DROP_PACKET - A router with a faulty link uses this command to redirect a data packet to *OsPhoenix*, which performs a packet rerouting (refer to Section IV.D).

IV. FAULT-TOLERANT MECHANISMS

This section describes the Phoenix fault tolerant mechanisms implemented in software, inside *OsPhoenix,* and in hardware, through *Fault monitor* and *Fault control machine*.

A. OsPhoenix Description

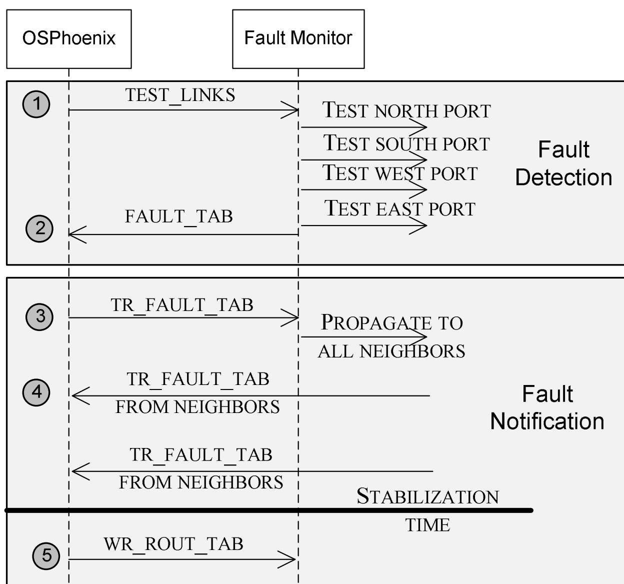

Figure 5 – Fault detection and notification mechanisms.

OsPhoenix is a small system placed into the PE's operating system that contains a *Global fault table* with the status of all NoC links, where the status informs if a given link was or not tested and, once tested, if it is a faulty link or not. *OsPhoenix* employs command codes to manage the *Fault control machine* and *Fault monitor*, performing four fault-tolerant mechanisms: (i) *Fault Detection*; (ii) *Fault Notification*; (iii) *Fault Reevaluation*; and (iv) *Packet Rerouting and Drop*. These mechanisms are described next. Additionally, Figure 5 details some steps of the two main fault mechanisms.

B. Fault Detection Mechanism

Phoenix fault model takes into account only faults on output ports of inter router links (i.e. links on ports NORTH, SOUTH, EAST and WEST). The *fault detection mechanism* starts with *OsPhoenix* sending to the local router the command TEST_LINKS (step 1 in Figure 5). Therefore, the *Fault Monitor* requests to the *Fault Control Machine* to send a predefined test packet to each output port. When the neighbor router receives a test packet, it loops back a packet with the same information. Therefore, the *Fault Monitor* is able to detect faulty links if one of the following conditions occurs, otherwise the link is considered tested and operating: (i) the low level control protocol fails; (ii) the test packet is not replied; or (iii) the test packet is replied but with different content.

Additionally, other routers identify a faulty router as a router containing faults in its entire inbound links eliminating the router from the possible routing paths. Finally, when *Fault Monitor* finishes all link tests it sends the *Faulty table* to *OsPhoenix* via FAULT_TAB command (step 2 in Figure 5).

C. Fault Notification Mechanism

The *fault notification mechanism* works in distributed way. Each *OsPhoenix* performs its *fault notification mechanism* propagating faults information to a neighbor *OsPhoenix* until it does not reach the stabilization condition.

Fault notification mechanism, which is accomplished by *OsPhoenix*, comprehends the following steps:

1. When *OsPhoenix* receives a packet with the *Fault table*, which can be from the local router (i.e. a FAULT_TAB command) or form another *OsPhoenix* (i.e. a TR_FAULT_TAB command), it verifies if its *Global fault table* is updated with the same values. If at least one value changed, *OsPhoenix* update the *Global fault table* and sends new packets with the same content (using only TR_FAULT_TAB command, step 3 and 4 in Figure 5) to all other neighbor *OsPhoenix*, otherwise *OsPhoenix* discards the received packet;
2. Since the beginning of the operation of *fault notification mechanism*, *OsPhoenix* starts a timer, which operates at same clock cycle of the NoC. This one is used to compare with the *Maximum Stabilization Time Period* (MSTP), which is the stabilization condition of the *fault notification mechanism*. When the timer reaches MSTP value, *OsPhoenix* considers that all faults where notified to all other *OsPhoenix*, enabling to compute the routing algorithm. When the algorithm finishes, *OsPhoenix* sends

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.

the WR_ROUT_TAB command (step 5 in Figure 5) containing the *Routing table* to the *Fault Control Machine*.

MSTP is proportional to the maximum NoC length, which is dependent on the NoC size and on the quantity and positioning of the faults. Since the quantity and positioning of the faults are not known during the design, we use here MSTP as the maximum NoC length delay, which is a worst case condition achieved by a packet passing through all routers.

D. Packet Rerouting and Drop Mechanism

Some NoC links may fail during Phoenix operation described in the previous two sections. These faults are taken into account similarly to the initial NoC operation. However, routers, which have not been updated with this fault information, consider that the fault does not exist and the router can transmit packets along the failed path. Aiming to avoid packets trapped inside the NoC due to a fault link, we implemented a *Packet rerouting mechanism* that runs on the router whose output port is defective. This mechanism redirects a packet, whose destination path passes through a fault link, to the local port, changing the data packet to a control packet (i.e. inserting the command DROP_PACKET). *OsPhoenix* reassemble the original data packet with a new path – a packet rerouting.

Figure 6 shows an example of the rerouting mechanism applied to a packet, which was sent from PE_A to PE_B following path 1. In this example, one link fail and the packet is directed to the local PE (i.e. PE_C) to be rerouted to path 3.

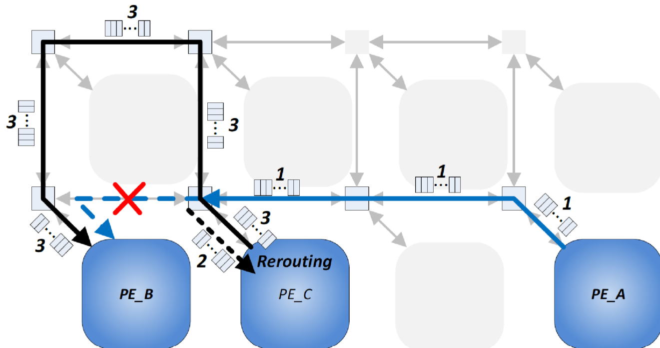

Figure 6 – Packet rerouting example.

When a fault occurs during the packet transmission (i.e. only part of the packet passes through the link and then the link fail) the *Packet rerouting mechanism* does not work properly. In fact, it was necessary to implement another solution that we call *Packet drop mechanism*: i) the flits that had already passed through the fault link compose an incomplete packet, which is propagated through all routers until reach the target PE. Then, the flits are discarded by *OsPhoenix*, since they compose an invalid packet; ii) the remaining packet flits are eliminated into the router. When a packet is dropped, the packet content is lost and it has to be resent at higher software levels.

E. Fault Reevaluation Mechanism

The *Fault notification mechanism* was designed to support only permanent faults. This approach facilitates the notification mechanism to stabilize quickly. However, the NoC was designed to support that a reevaluation of the faulty status of all NoC links, which any *OsPhoenix* may require in a distributed manner using RST_ALL_FAULTS command. This procedure guarantees that transient faults are not accidentally detected as a permanent fault.

The reset command is sent to all neighbor PEs, which upon receiving this command, they: (i) reset their *Global fault table*; (ii) request to their local router to reset the respective *Fault table* through RST_FAULT_TAB command; and (iii) retransmit the RST_ALL_FAULTS command to their respective neighbor PEs. During MSTP time the NoC stops running again, thus this time is used to propagate the system reboot message. After MSTP, the operation is exactly equal to the one performed at startup, i.e. a fault detection followed by a fault notification.

V. EXPERIMENTAL RESULTS

This section demonstrates the stabilization time of the distributed fault notification mechanism.

Figure 7 illustrates the total stabilization for zero to 16 simultaneous faults and NoCs of different sizes. The results demonstrate that the number of faults typically has a small effect on the stabilization time. The exception is when the number of faults is so large that most of the NoC links are faulty. This is the case with 16 faults on a 4×4 NoC. This NoC has 24 links and 16 of them are faulty. In this situation the stabilization time decreases since the notification process has to evaluate fewer links. Remark that for a square mesh NoC with side L the number of links (n_L) is computed with equation $n_L = 2 \times (L^2 - L)$.

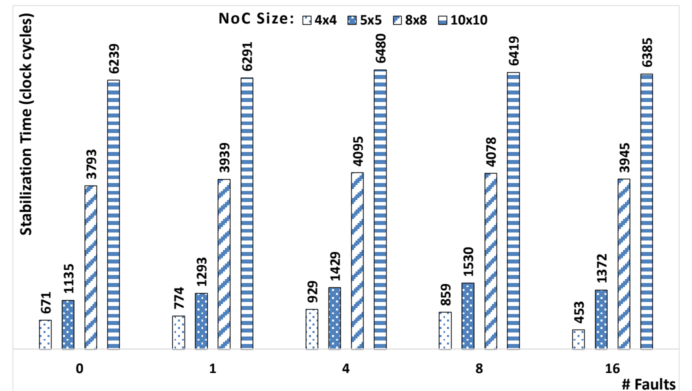

Figure 7 – Total monitoring stabilization time per number of faults considering four NoC sizes.

Figure 8 highlights the previously mentioned effect since in this chart the number of NoC routers normalizes the stabilization time. For instance, if the total stabilization time takes 700 clock cycles on a 4×4 NoC (i.e. 16 routers), then 700/16 is the normalized value.

This chart shows that the stabilization time presents low rate with low quantity of fault links, due to the absence of faults implies many paths with less hops to propagate the fault notification. When the quantity of faulty links increases, the stabilization time also increases. On the other hand, when the quantity of faulty links becomes large compared to the total quantity of links, the stabilization time decreases because faulty links probably split the NoC, reducing the communication paths. Additionally, the stabilization time per router is around 55 clock cycles for NoCs of different sizes.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.

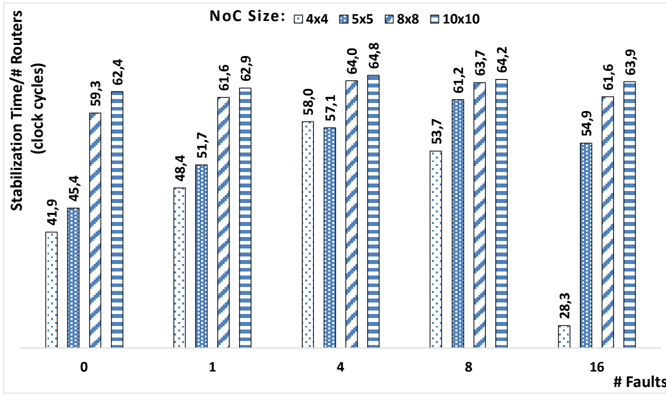

Figure 8 – Stabilization time normalized by the number of routers of the NoC.

Figure 9 and Figure 10 illustrate the scalability of the proposed approach for a scenario where the NoC is dedicated to a traffic containing only monitoring packets, i.e. it does not contain packets of data.

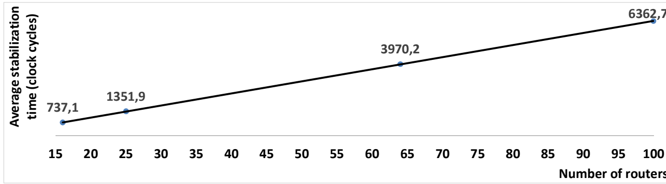

Figure 9 – Monitoring stabilization time for some sizes of square NoCs.

Figure 9 shows the stabilization time of 4 square NoCs (i.e. 4×4, 5×5, 8×8, 10×10) considering that all links are operating without faults. The results state that the stabilization time of the proposed approach is practically linear with respect to the quantity of routers.

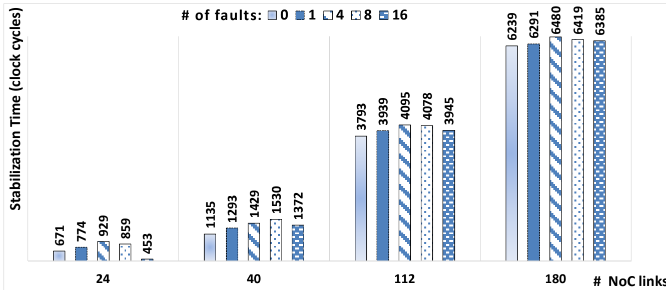

Figure 10 – Monitoring stabilization time normalized by the number of NoC links.

Figure 10 depicts that the stabilization time is also linear related to the number of NoC links (i.e. n_L) and the number of faults does not have significant influence.

VI. CONCLUSION

Large MPSoCs are becoming more demanding in terms of architecture design to meet the ever-rising application requirements at runtime. Moreover, in this context, fault diagnosis is crucial in order to guarantee system recovery and reduced latency in case of faults. One alternative to overcome this problem is applying fault-tolerant mechanisms in order to recover from faults and discover alternative paths in the network. To address this need, this work presents Phoenix, which is a fault-tolerant 2D mesh NoC that enables properly communication in case of finding manufacturing faults, or most importantly, faults occurred during execution.

The propagation of faults is an iterative process, where neighbors relay the fault notifications to others until reaching a stabilization condition. The fault notification mechanism uses a maximum period as stabilization condition, since it is proportional to the maximum NoC length and copes with runtime faults. Experimental results show that Phoenix is scalable in terms of stabilization time and allows the MPSoC operation even in the presence of several faults.

ACKNOWLEDGMENT

This work is partially funded by FAPERGS PqG 12/1777-4 and Docfix SPI n.2843-25.51/12-3. Financial support also granted by CAPES AEX 5967-13/9, CNPQ and FAPESP to the INCT-SEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9.

REFERENCES

[1] A. Jantsch and H. Tenhunen. Network on Chip. Kluwer Academic Publishers, 312p., Jan. 2003.

[2] É. Cota, A. Amory and M. Lubaszewski. **Reliability, Availability and Serviceability of Networks-on-Chip**. *Springer*, 2012, p. 209.

[3] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw and D. Sylvester. **Vicis: a reliable network for unreliable silicon**. *Design Automation Conference*, pp. 812–817, 2009.

[4] D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, and D. Blaauw. **A highly resilient routing algorithm for fault-tolerant NoCs**. *Design, Automation, and Test in Europe*, pp. 21–26, 2009.

[5] Z. Zhang, A. Greiner and S. Taktak, **A reconfigurable routing algorithm for a fault-tolerant 2D-Mesh Network-on-Chip**. *Design Automation Conference*, pp. 441–446, 2008.

[6] E. Wachter, A. Erichsen, A. Amory and F. Moraes. **Topology-Agnostic Fault-Tolerant NoC Routing Method**. *Design, Automation, and Test in Europe*, 2013.

[7] C. Feng, Z. Lu, A. Jantsch, J. Li and M. Zhang, **A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for network-on-chip**. *Proceedings of Workshop on Network on Chip Architectures - NoCArc*, pp. 11–16, 2010.

[8] A. Mejia, M. Palesi, J. Flich, S. Kumar, P. Lopez, R. Holsmark and J. Duato, **Region-Based Routing: A Mechanism to Support Efficient Routing Algorithms in NoCs**. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 3, pp. 356–369, Mar. 2009.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 11,2022 at 20:08:07 UTC from IEEE Xplore. Restrictions apply.