A Fast Runtime Fault Recovery Approach for NoC-Based MPSoCS for Performance Constrained Applications

Eduardo Wachter, Augusto Erichsen, Leonardo Juracy, Alexandre Amory, Fernando G. Moraes
FACIN – PUCRS – Av. Ipiranga 6681– Porto Alegre – RS – Brazil
{eduardo.wachter, augusto.erichsen, leonardo.juracy}@acad.pucrs.br, {alexandre.amory, fernando.moraes}@pucrs.br

Abstract - *Mechanisms for runtime fault-tolerance in Multi-Processor System-on-Chips (MPSoCs) are mandatory to cope with transient and permanent faults. This issue is even more relevant in nanotechnologies due to process variability, aging effects, and susceptibility to upsets, among other factors. The literature presents isolated solutions to deal with faults in the MPSoC communication infrastructure. In this context, one gap to be fulfilled is to integrate all layers, resulting in a solution to cope with NoC faults from the physical layer up to the application layer. The goal of this work is to present a runtime integrated approach to cope with NoC faults in MPSoCs. The original contribution is the proposal of a set of hardware and software mechanisms to ensure both efficient and reliable communication in NoC-based MPSoCs. The proposal has an acceptable silicon area overhead and a small memory footprint. Experiments demonstrate that benchmarks (synthetic and real MPSoC applications) were simulated with thousands of random fault injections, and all of them were executed correctly. Moreover, the average application execution time overhead is lower than 0.5%. This suggests the proposed fault tolerant method could be used in applications with reliability and performance constraints.*

Keywords - *NoC-based MPSoC, fault-tolerant NoCs, fault-tolerant communication, fault recovery.*

I. INTRODUCTION

The *productivity gap* refers to the fact that the number of available transistors grows much faster than the ability to design and integrate them. One solution to deal with the *productivity gap* is the reuse of components, resulting in complex SoCs. MPSoCs are SoCs that reuse processing elements, interconnecting them through a bus or a Network-on-Chip (NoC). Due to scalability reasons, NoCs are the infrastructure used in larger MPSoCs. Despite the possibility to integrate in the near future thousand of cores on a chip [1], process variability, and aging effects become prominent in nanotechnologies, leading to soft-errors and permanent faults.

The adoption of NoC-based MPSoCs provides an excellent opportunity for the design of reliable systems. For example, there are redundant paths in the NoC and redundant processors. These features enable a graceful degradation of the MPSoC in the presence of faults, by keeping its functionality intact at the expense of some acceptable performance penalty [2].

This work proposes a layered approach for runtime fault recovery in NoC-based MPSoCs. At the router layer, the designer uses any appropriate test method (functional, BIST, ECC) to detect fault ports. The test just needs to trigger the proposed method, which is built on top of a small specialized network used to search fault-free paths, and an MPI-like software protocol, implemented at the OS level, hidden from the application layer. The software protocol detects unresponsive routers, and automatically fires the path search method, which can quickly return a new path to the source processor. This approach has a distributed and scalable nature, since a source processor uses only local information to monitor the path to its targets. In addition, the proposed approach provides complete reachability, i.e. if there is only one faulty-free path to the target processor, it will be found, even in the presence of multiple faults.

The rest of this paper is organized as follows. Section II reviews related works to fault tolerant NoCs. Section III presents the main MPSoCs architectural features, as well as the adopted communication protocol. Section IV details the FT NoC. Section V integrates the fault recovery protocol to the FT NoC, corresponding to the main contribution of the present paper. Section VI presents the results, focusing on the FT communication behavior. Section VII concludes this paper.

II. RELATED WORKS

This Section reviews recent works related to: (*i*) fault tolerance (FT) at the link level [4][5]; (*ii*) FT at the router level [6][7][8]; (*iii*) FT at the routing algorithm level [9][10]; (*iv*) FT at the communication API level [11][12][13]. The Radetzki's survey [3] contains a comprehensive state-of-the- art in FT methods for NoCs.

Veiga and Zeferino [4] use two fault tolerant techniques at the link level: CRC and parity. In both cases, the error is detected at the flit level, requiring flit retransmission. The work targets transient faults, due to crosstalk errors. The Authors report an area overhead between 6.25% (parity) to 28% (CRC). This approach is not coupled to a system-wide fault recovery solution. Ghossal and Das [5] present a network topology using long links between routers. The goal of the technique is to reduce the network latency, increase parallelism, and improve the traffic distribution. Their approach is coupled to an FT routing algorithm. According to the Authors, the proposed technique tolerates only some specific types of faulty situations.

Chang et al. [6] propose a technique with redundant routers to increase reliability. The technique corrects a permanent fault for each NoC column, which can occur at the routers or links. The technique consists of adding a router at the top of each column of a 2D-mesh NoC. When a fault is detected, the NOC is reconfigured, without changing the original topology. This method supports a limited number of multiple faults since there are a limited number of spare routers per column. Tsai et al. [7] propose the Bidirectional Fault-Tolerant NoC (BFT-NoC) able to mitigate both static and dynamic channel failures. The BFT-NoC uses bidirectional channels to provide fault-tolerance capability instead of using FT routing algorithms. A faulty link or port induces a router reconfiguration, changing the healthy link to a bi-directional link. Multiple faults at the adjacent links can break the method. Fick et al. [8] proposes the Vicis NoC, which uses the inherent redundancy in the network and its routers to maintain correct operation. Each router has a BIST that diagnoses the locations of permanent faults and uses a port swapping approach (when a given port fails, it is possible to use a neighbor port to receive the packets), and a crossbar bypass bus to mitigate the fault.

Rodrigo et al. [9] present the uLBDR, a router architecture for fault-tolerant routing in NoCs. It stores the connectivity status of the neighbors' routers, leading to minimal paths. The uLBDR proposes the replication of the packets in two output ports, ensuring complete reachability but increasing network congestion. Alhussien et al. [10] present an FT routing where the NoC has two channels in the horizontal direction (north and south) and one channel in the vertical direction (east and west). The routing guarantees absence of deadlocks and livelocks up to two faulty channels. Based on the faults in the network, alternative routes are used to reroute packets.

Aulwes and Daniel [11] include reliability features on MPI such as checksum, message retransmission, and automatic message re-routing. Kariniemi and Nurmi [12] present a fault-tolerant communication library targeting MPSoCs based on packet retransmission, where watchdog timers and CRCs are used to detect packet losses and bit errors caused by transient faults at the NoC links. Both [11] and [12] can only be applied on applications with very regular traffic patterns; otherwise, false fault alerts can be fired. Hébert et al. [13] proposes a distributed fault-handling method for a NoC-based MPSoC. Faults at the PE are detected with heartbeat messages exchanged between PEs. Each PE has a watchdog timer capable of detecting missing or corrupted heartbeat messages. Besides fault detection, the method also comprises the isolation or the faulty PE and recovering via task remapping. As stated at the conclusion of [13], the approach does not handle faults at the interconnect network.

The reviewed works improve the reliability of a NoC-based MPSoC at a limited extent. Most reviewed approaches either support limited number of NoC faults [4-10] or work for limited applications [11,12]. Moreover, either the approach is tightly coupled to a specific NoC test method [8,9], or the test method is not complete.

The *contribution* of the present work lies on a layered fault tolerant method for MPSoCs. The test layer is not coupled to a specific NoC test method, so the designer can choose the most appropriate test approach for the application. At the reconfiguration layer, the proposed path recovery mechanism is very fast and works with multiple NoC faults. At the application layer, the application source code is not changed. It is worth mentioning that this work focuses in the NoC fault reconfiguration layer. NoC test methods and fault tolerant PEs are not addressed in this paper.

III. MPSOC COMMUNICATION MODEL

An MPSoC consists of a set of PEs interconnected by a given network topology. This paper adopts common features found in MPSoCs [14]:

(i) each PE contains one processor (32 bits MIPS) with a private memory (64KBytes), DMA controller, Network Interface (NI), and the router described in Section IV;

(ii) homogeneous processing, i.e., all PEs have the same architecture. The software part of the FT approach is entirely written in C, thus, it can be compiled to a different target processor, supporting heterogeneous MPSoCs. The restriction is only related to our experiment, which uses MIPS-based processors;

(iii) applications are modeled as task graphs;

(iv) communication model is message passing;

(v) a mapping function maps tasks onto PEs, being possible to have more than one task per PE (multi-tasking operating system (OS)).

The communication API adopts two MPI-like primitives: a non-blocking *Send()* and a blocking *Receive()*. To implement a non-blocking *Send()*, a dedicated memory space in the OS, named *pipe*, stores each message written by tasks. Two main reasons justify the adoption of such protocol:

(i) a packet is injected into the NoC only when it is requested. This feature reduces NoC congestion, because a given PE is waiting the packet for consumption and the injection in burst mode avoid idle time between flits;

(ii) packet retransmission has low latency penalty since packets are buffered in the processor local memory before transmission. Therefore, just the network latency is accounted, which is typically lower than one hundred of clock cycles. Moreover, end-to-end error control may be used with no extra hardware cost.

Figure 1 details the communication protocol used for data packets, i.e., packets exchanged between tasks. The execution of a *send*() transfers the contents of the message to a free *pipe* position, assigning its status to *USED*. When task B (assumed mapped in PE_2) executes a *receive*(), a packet with *a message request* is sent to PE_1. Note that task B is now blocked, in a *WAITING* state. The reception of a *message request* by PE_1 results in two actions: (*i*) the pipe status changes to *WAITING_ACK* (1 in Figure 1); (*ii*) the message is delivered to PE_2 (2 in Figure 1). PE_2 transfers the message contents to task B memory space, enables its execution (status READY), and sends an acknowledgement packet (3 in Figure 1) to PE_1. Upon acknowledgement reception, the pipe position in PE_1 is released (4 in Figure 1).

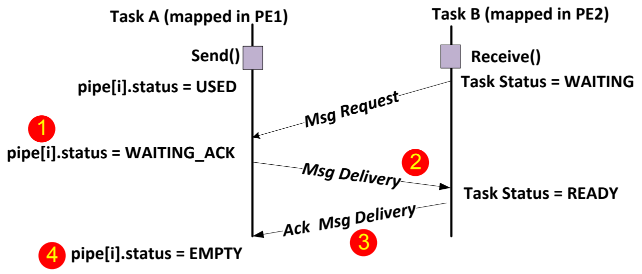

Figure 1 – Communication protocol to exchange data packets.

The communication protocol has another two features related to reliability. The first one is that a packet can only be delivered, i.e. injected into the NoC, if there is no *pipe* position in *WAITING_ACK* status. This feature avoids the injection of a packet without the reception acknowledgement of the previous injected packet. Also, each packet receives a unique sequence number. This feature avoids the reception of the same packet twice, due to packet retransmission.

IV. FAULT TOLERANT NOC MODEL

This work adopts the following common features found in NoCs [15]: 2D-mesh topology; wormhole packet switching; adaptive routing algorithm; duplicated physical channels. Features related to fault tolerance include:

(i) simultaneous support to distributed and source routing;

(ii) isolation wrapper cells at the NoC's input ports;

(iii) presence of a secondary path discover network (PDN).

This section initially introduces the proposed method, which is detailed in the next subsections. The entire process starts when a fault is detected in the network by any fault detection method at disposal in the NoC design. The fault detection causes the faulty region (a single port or even the entire router) to be closed to its neighbor routers. This is performed by isolation wrapper cells (i.e. conventional test wrapper cells used for core test wrappers) present at all input ports. When a given port is isolated by these cells, all packet directed to the port are automatically dropped to avoid generating a corrupted packet. In addition, the neighbor port that sends the packet to the blocked port generates a *packet retransmission request* back to the source PE via the PDN, not the main network.

If the fault is detected in the middle of the packet transmission, the isolation cells discard part of the packet, and the other part of the incomplete packet arrives at the target PE. In this case, the OS of the target PE discards this incomplete packet and request retransmission.

When the packet retransmission request reaches the source PE, it fires the *path discovery process*, which returns a fault-free path to the target if this path exists. However, this path might generate deadlocks. Fully adaptive routing is granted using two disjoint networks, using duplicated physical channels [16]. Physical channel 0 routes packets according to the west-first routing algorithm, and physical channel 1 according to the east-first routing algorithm. In the absence of faults, the routers use determinist XY routing algorithm. The XY turns are a subset of the west- and east-first routing algorithms, ensuring the absence of deadlocks [16]. Thus, the source PE analyses the suggested fault-free path for invalid turns that might cause deadlocks. These invalid turns are simply avoided by switching physical channels. Therefore, there are no cycles, avoiding deadlocks. All the subsequent transmissions to this target PE use this new fault-free and deadlock-free path.

Figure 2 shows a simplified view of the NoC router, without the presence of the PDN. The test wrapper cells (see TW) in Figure 2 signalize to neighbor routers a fault in the input port (see *f* signal).

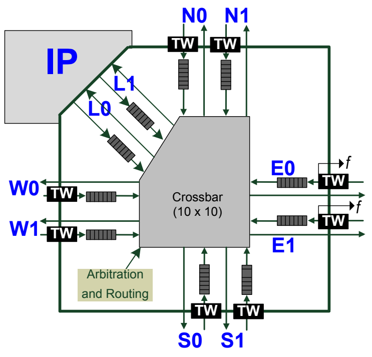

Figure 2 – Router model adopted in the present work. All test wrappers (TW) signalize to the neighbor routers if the input port presents fault, as illustrated in ports E0 and E1 by signal *f*.

A. Path Discover Network

The *path discover network* (PDN) is a simple network, responsible for executing actions related to fault tolerance. It is employed as a parallel network, but integrated to the router, to avoid interference with the traffic of data packets.

Each PDN router (Figure 3) communicates with other PDN routers using the following set of signals:

- tuple {R_S, R_T, hop#}, corresponding to the addresses of the source PE, target PE, and the number of hops from the source PE;
- *out_seek/in_seek*, carries the service that should be executed by the PDN;
- *in_ack_seek*/*out_ack_seek*, the acknowledgement of the requested service.

The first service executed by the PDN network is named *SEEK* (*out_seek=SEEK*). The function of the *SEEK* service is to propagate a "wave" from a given PE to another PE, as illustrated in Figure 4 (idea borrowed from old maze routing algorithms). When a *SEEK* is received, the port that received the service and the tuple values are stored in the *seek_table* (small table with four entries). Then, each PDN router sends to all neighbors the tuple {R_S, R_T, hop#+1}, excepted to the one originating the *seek_request*. In this way, the wave is propagated to all PDN routers. If a *seek_request* is received in a given port, and R_S is already stored in the *seek_table*, the request is discarded. This ensures that the wave will be propagated to all routers, and when all routers were visited the process stops without any complex control. In both cases, "store/propagate seek" and "discard seek", an acknowledgment is sent to the PDN router that requested the wave propagation.

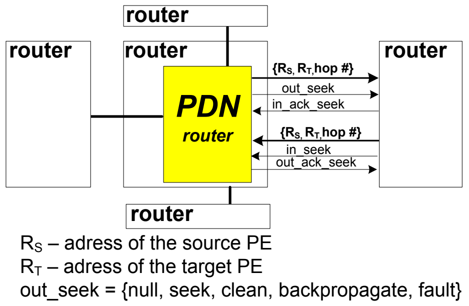
router
router
router
router
router
PDN
router
{RS, RT,hop #}
out_seek
in_ack_seek
{RS, RT,hop #}
in_seek
out_ack_seek
RS – adress of the source PE
RT – adress of the target PE
out_seek = {null, seek, clean, backpropagate, fault}

Figure 3 – PDN router, with its external interface.

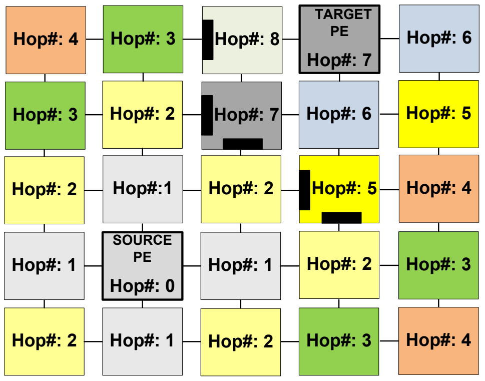

Figure 4 – "Wave" propagation in the PDN. Black square rectangles correspond to faulty ports.

Note that in Figure 4 some input ports are assumed faulty. Each router port has a control bit, named *faulty_port*, where the test method must set in case it detects faults in the port under test (see signal *f* in Figure 2). Whatever the test method used in the design, it must only set this *fault_port* bit when it detects faults. This action triggers the proposed fault recovery method. Even with multiple faulty ports, the wave visits all NoC routers. The hop# in the Figure corresponds to the minimal distance between R_S and R_T (7 hops), considering this fault scenario.

The second PDN service is the *CLEAN* process (*out_seek=CLEAN*). It is similar to the *SEEK* service, in the sense that it propagates a wave to all routers. The goal of this service is to clean the fields in *seek_table* used by the wave. R_T injects in the PDN network the same received tuple {R_S, R_T, hop#}, and all PDN routers with this entry in the *seek_table* with R_S have to free this entry.

The third service is named *BACKPROPAGATE*. R_T injects a *backtrack packet* into the NoC (not in the PDN) to R_S, with the payload containing the ports stored by each PDN router to reach R_T. The *backtrack packet* uses source routing, following the reverse path followed by the seek step. The *BACKPROPAGATE* service ends when R_S is reached. The backtrack packet is transferred to the processor to check the validity of the path. To be able to execute any turn in the path without deadlocks, a combination of east-first and west-first routing algorithms is used. It is assumed west-first in channel 0 and east-first in channel 1. The OS is responsible for assigning the turns/channels in the path.

The fourth service, *FAULT*, is transmitted to R_S when a packet hit a faulty port. This service is also similar to the first one, *SEEK*, i.e., a wave is propagated to all routers using the PDN. The difference between the *SEEK* and the *FAULT* services is that the first one searches a new faulty-free path to a given PE, and the second one informs R_S that the current path became faulty.

B. Summary of PDN's Features

The baseline router with duplicated physical channels and the modified router including the PDN were synthesized to Xilinx FPGA (xc5vlx330tff1738). The baseline uses 494 flip-flops and 2226 LUTs while the router with the PDN uses 726 flip-flops and 2913 LUTs. This is equivalent to an area overhead of 47% for flip-flops, and 30% for LUTs compared to the baseline router. Moreover, the PDN Router's area is constant despite the network size, thus, the hardware is scalable.

This is a small cost considering the increased NoC robustness and considering that the router area is much smaller than the processor and memory area. Experiments show that the router silicon area is typically less than 10% of the entire PE area. The intrusiveness in the NoC traffic is minimal because only backtrack packets effectively use the NoC. Finally, the hardware implementation of the wave propagation (used by the SEEK, CLEAN, and FAULT services) ensures low effect on application's performance.

V. FT COMMUNICATION PROTOCOL

At the NoC level, it is possible to adopt two main strategies to cope with faults: router-to-router or end-to-end [19]. Router-to-router methods are suitable for transient faults, requiring flit retransmission or error correction codes. End-to-end methods require packet retransmission, but may be used with transient and permanent faults. This work adopts the *end-to-end* strategy, since our goal is to support transient and permanent faults.

An FT communication requires a complete stack of layers to ensure reliable communication. Such stack includes:

- test method for input ports or the entire router;
- wrapper cells to isolate faulty regions and to signalize faults;
- two physical channels with adaptive routing to cope with faulty paths;
- support at the OS level to compute new paths, to retransmit packets, and to discard incomplete packets.

Figure 5 details the FT complete communication protocol with path reconfiguration and message retransmission. As explained in Section III, the execution of *Send()* by a given task transfers the message to the OS level, into the *pipe* structure. The task requesting data injects a *message request* in the NoC (1 in Figure 5). Then, the OS transmits the message, through a *message delivery* packet (2 in Figure 5). In this Figure, the message reaches a faulty router (fault in the west input port). The last faulty-free router, P_{ff}, fires the fault recovery process.

The P_{ff} router discards all received flits in the affected port, and uses the PDN to start a wave from P_{ff} (*FAULT* service) to R_s (PE1) (3 in Figure 5). This step is managed locally at the router, does not involving the processor. Therefore, the wave reaches R_s in few clock cycles.

In Figure 5, step 4, when the FAULT wave reaches PE1, the processor is interrupted, and two actions are managed by the OS: (*i*) the PDN launches a CLEAN wave to free the slots of the *seek_table* used by the *FAULT* wave; (*ii*) the PDN launches a *SEEK* wave to reach R_T (PE2).

In Figure 5, step 5, the NoC is used to transfer the backtrack packet. At each hop, the packet receives the port identification from which the *SEEK* wave entered. For example, in Figure 4, the backtrack packet could be E-E-E-N-N-N-W (3 hops to East, 3 hops to North, 1 hop to West).

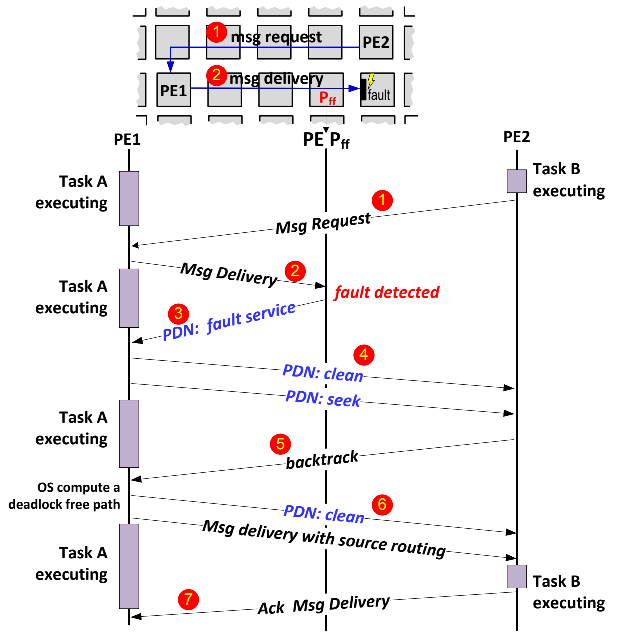

Figure 5 – FT communication protocol.

In Figure 5, step 6, when the backtrack packet reaches PE1, the PDN launches a CLEAN wave to free the slots of the *seek_table* used by the *SEEK* wave, and the OS computes a deadlock free path. Using the previous example, there are two turns in the path: E-N and N-W. The first turn is allowed in the west-first algorithm, but not the second one. The resulting path is then E0-E0-E0-N0-N0-N0-W1. This means that the first 6 hops use channel 0, and the last hop channel 1, breaking a possible deadlock cycle. This path is stored in the OS, and all subsequent communications between PE1 and PE2 use this new path based on source routing. The software complexity to generate the deadlock free path grows linearly with the number of hops in the path.

In Figure 5, step 7, the target PE receives the message, schedules task B to execution, and sends the acknowledgment required by the protocol. Finally, the source PE can release the message from the pipe.

Even though Figure 5 illustrates a fault in the message delivery part of the protocol, the faults may happen at any step of the protocol, e.g., during the message request or the acknowledgment delivery. In any case, the OS is able to identify the lost message, trigger the path reconfiguration and resend the message.

A. Summary of the Proposed FT Protocol

Using scenarios considered as worst-cases, with several faulty configurations, and a large number of hops between R_s-P_{ff}, and P_{ff}-R_T the message delay increased at most 3 times (section I.A). This is a remarkable result, since:

- watchdog timers, which is the most common approach to detect faulty paths [11][12], are not necessary to detect that messages were not delivered. Watchdog timers induce penalties in order of 100's of thousand of clock cycles, since messages may be delayed due congestion or faults. Watchdog timers should wait enough time to avoid false fault detections. Experiments with watchdogs timers instead the proposed protocol resulted in a worst-case execution time overhead of 65%, and there is no guarantees that false fault alerts were not generated [18].
- small performance penalty, as presented later, since only one message is delayed. The message penalty corresponds to the FT protocol and the time required to retransmit the message.

- small memory footprint, the protocol software design increased the OS size in 6.1 KB.
- small and *scalable* hardware (PDN) and software (path computation, control of the PDN) complexity. The PDN hardware is scalable because it is independent of the network size, and the software is scalable due to its distributed nature with no global information required and linear complexity.

VI. RESULTS

A. *Protocol Evaluation*

Table 1 presents the performance of the FT communication protocol (Figure 5), varying the distance between the source processor (PE_1) to the target processor (PE_2). A distance superior to 7 hops is not realistic, since mapping heuristics has as cost function to map communicating tasks as close as possible. The application is an MPEG decoder, with 128-word messages (256-flit packets) from PE_1 to PE_2. There is no congestion in the path, since the goal is to characterize the protocol. In Table 1:

- *t2→t3 – time required to inject a packet in the network, and the* reception of the fail detection. To inject a packet in the network, the OS: (*i*) configures the DMA module to transmit the packet; (*ii*) restores the context of the current task; (*iii*) enables external interrupts to be received. Meanwhile, the fail detection interruption is received. As the time spent in the NoC is smaller than the number of cycles consumed by the OS, this parcel of the protocol is constant regardless the number of hops.
- t3→t4 – time spent between the fault detection by PE_1 and the reception of the *SEEK* by PE_2. The OS in PE_1 is interrupted by the fail detection packet. It triggers a clean wave to empty the fault detection and then triggers a seek to PE_2. Its typically increases 16 clock cycles per hop.
- t4→t5 – time to transmit the backtrack packet. Due to the hardware implementation, a small number of clock cycles is observed, and it is a function of the number of hops.
- t5→t6 – corresponds linearly to the number of cycles to compute the new path in the OS.
- t6→t7 – time spent to deliver the packet using the faulty free path. The number of clock cycles is constant for the same reason explained in the parcel t2→t3.

Table 1 - Number of clock cycles for each step of the FT communication protocol of Figure 5 varying the number of hops between PE_1 and PE_2. t2 to t7 represent the time spent in each step of the proposed protocol.

hops #	t2→t3	t3→t4	t4→t5	t5→t6	t6→t7	**TOTAL**
2	769	647	48	981	1023	**3468**
3	769	672	51	1084	1023	**3599**
4	769	688	63	1177	1023	**3720**
5	769	704	75	1291	1023	**3862**
6	769	720	87	1402	1023	**4001**
7	769	736	99	1499	1023	**4126**

In these scenarios, the amount of time to transmit a packet in a fault-free scenario is typically 1670 clock cycles (Figure 6). Therefore, the worst-case overhead is observed in the seven hops scenario, corresponding to 2.46 times the time to transmit a message without faults. The transmission of the first message (Figure 6) is faster, because it is buffered while the consumer task starts its execution. The third message requires path reconfiguration and message retransmission. All other messages are not penalized since the new path is also minimal, and the time to transmit a message in both cases is the same. Consequently, the impact of the fault-detection/path reconfiguration/retransmission in the application execution time is minimal, as will be presented in the next subsection.

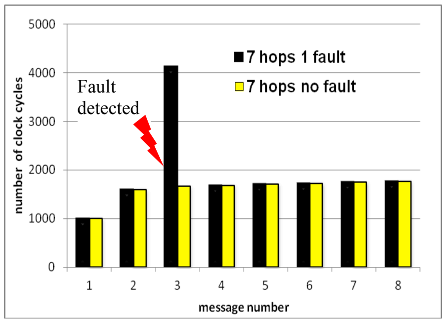

Figure 6 – Time spent to transmit eight 256-flit packets with and without fault using the proposed FT communication protocol for the 8 first frames. The fault was detected in the third packet.

B. *Execution Time Evaluation*

Three applications are used as benchmarks to evaluate the proposed approach: one synthetic application (*synthetic*), used for validation purposes; two real applications, DTW (Dynamic Time Warping) and MPEG. These three applications were decomposed into communicating tasks.

A random fault insertion campaign is used to validate different fault configurations. Faults are injected in the router ports, excepting the local port, because this paper assumes NoC faults only. An automatic fault analysis flow was created, which includes the MPSoC generation, the creation of fault scenarios, simulation of all scenarios, and performance evaluation.

For each application, 1,200 scenarios are simulated (MPSoC modeled at the RTL level with VHDL). Two faults are injected at each scenario at the same moment, at random ports. The simulations demonstrate that the method supports several simultaneous faulty ports, as long as the number of faults does not physically split the application graph into disjoint graphs. Figure 7 presents the execution time for the 3,600 simulated scenarios.

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

- The *synthetic* application represents an extremely pessimistic application where there is no computation, just communication. Thus, this is not realistic compared to actual applications where, assuming a good application partition, it is expected to spend more time with computation than communication. Its worst-case execution time overhead is 6.5%, which corresponds to 0.08 ms.
- Independence of the fault location. The overhead is mostly induced by the *moment* of the fault detection. If the PE is executing some OS function, the treatment of the interruption due to the fault may be delayed. Thus, the variation observed in the results is mostly induced by the OS's task scheduling scheme.
- Fast reconfiguration. As presented in Table 1, the impact due to the hop number to reconfigure the path and retransmit the packet is negligible compared to the application execution time and OS's scheduling time.
- The packet latency is only affected in the undelivered message. After path reconfiguration, there is no overhead in the communication.

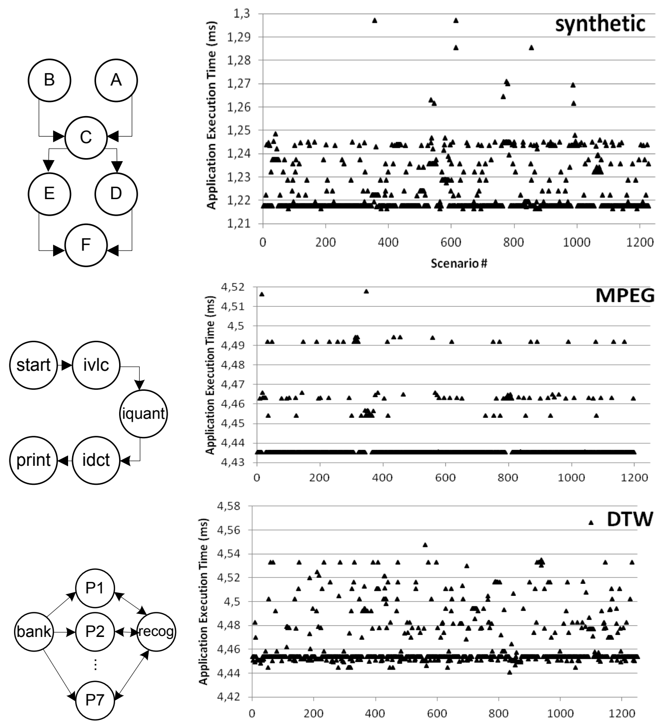

Figure 7 – Applications' task graphs, and the execution time for each faulty scenario.

A final remark is that all 3,600 simulated scenarios finished the applications correctly. This fact means that the proposed method was able to treat the injected faults, with minimal impact in the execution time and no impact on the application functionality.

VII. CONCLUSION

As far as the Authors could verify, this is the first work integrating all required layers to cope with NoC faults, ensuring fast and correct execution of applications without modifying its source code. NoC faults are handled at runtime, with negligible performance penalty on average case. Such small performance penalty is the result of the use of a small and fast network (PDN) to seek faulty free paths. In addition, the proposed method is scalable, and the PDN does not disturb the applications' traffic. These features and costs suggest that the proposed method may be used for applications with thigh reliability and performance constraints.

ACKNOWLEDGEMENTS

The author Fernando Moraes acknowledges the support granted by CNPQ, processes 472126/2013-0 and 302625/2012-7; and CAPES process 708/11.

REFERENCES

[1] Borkar, S. *Thousand Core Chips - A Technology Perspective*. In: DAC, 2007, pp.746-749.

[2] Rodrigues, R.; Kundu, S. *On graceful degradation of chip multiprocessors in presence of faults via flexible pooling of critical execution units*. In: IOLTS, 2011, pp. 67-72.

[3] Radetzki, M.; Feng, C.; Zhao, X.; Jantsch, A. *Methods for Fault Tolerance in Networks on Chip*. ACM Computing Surveys, v.46(1), 2013, article No. 8.

[4] Veiga, F.; Zeferino, C. *Implementation of Techniques for Fault Tolerance in a Network-on-Chip*. In: WSCAD-SCC, 2010, pp. 80-87.

[5] Ghosal, P.; Das, T. *FL2STAR: A Novel Topology for On-Chip-Routing in NoC with Fault Tolerance and Deadlock Prevention*. In: CONECCT, 2013, 6p.

[6] Chang, Y.; Chiu, C; Liu, S.; Liu, C. *On the Design and Analysis of Fault Tolerant NoC Architecture Using Spare Routers*. In: ASP-DAC, 2011, pp. 431-436.

[7] Tsai, W.; Zheng, D.; Chen, S.; Hu, Y. *A Fault-Tolerant NoC Scheme Using Bidirectional Channel*. In: DAC, 2011. pp. 918-923.

[8] Fick, D.; DeOrio, A.; Jin Hu; Bertacco, V.; Blaauw, D.; Sylvester, D. *Vicis: A Reliable Network for Unreliable Silicon*. In: DAC, 2009, pp. 812-817.

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

[10] Alhussien, A.; Bagherzadeh, N.; Verbeek, F.; van Gastel, B.; Schmaltz, J. *A formally verified deadlock-free routing function in a fault-tolerant NoC architecture*. In: SBCCI, 2012, 6p.

[11] Aulwes, R.; Daniel, D.; Desai, N.; Graham, R.; Risinger, L.; Taylor, M.; Woodall, T.; Sukalski, M. *Architecture of LA-MPI, a Network-Fault-Tolerant MPI*. In: IPDPS, 2004, 6p.

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

[13] Hébert, N.; Almeida, G.; Benoit, P.; Sassatelli, G.; Torres, L. *Evaluation of a Distributed Fault Handler Method for MPSoC*. In: ISCAS 2011, pp. 2329–2332.

[14] Garibotti, R.; Ost, L.; Busseuil, R.; Kourouma, M.; Adeniyi-Jones, C.; Sassatelli, G.; Robert, M. *Simultaneous Multithreading Support in Embedded Distributed Memory MPSoCs*. In: DAC, 2013, 7p.

[15] Agarwal, A.; Iskander, C.; Shankar, R. *Survey of Network on Chip (NoC) Architectures & Contributions*. Journal of Engineering, Computing and Architecture, v.3(1), 2009.

[16] Young J.; Concer, N.; Petracca, M.; Carloni, L. *Virtual Channels vs. Multiple Physical Networks: A Comparative Analysis*. In: DAC, 2010, pp. 162-165.

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

[18] Wachter, E.; Erichsen, A.; Juracy, L.; Amory, A.; Moraes, F. *Runtime fault recovery protocol for NoC-based MPSoCs*. In: ISQED, 2014, pp.132-139.

[19] Murali, S.; Theocharides, T.; Vijaykrishnan, N.; Irwin, M.J.; Benini, L.; De Micheli, G. *Analysis of Error Recovery Schemes for Networks on Chips*. IEEE Design and Test of Computers, v.22(5), 2005, pp. 434-442.