Runtime Fault Recovery Protocol for NoC-based MPSoCs

Eduardo W. Wächter, Leonardo R. Juracy, Walter L. Neto, Alexandre M. Amory, Fernando G. Moraes
FACIN – PUCRS University – Av. Ipiranga 6681– Porto Alegre – RS – Brazil
{eduardo.wachter, leonardo.juracy, walter.lau}@acad.pucrs.br, {alexandre.amory, fernando.moraes}@pucrs.br

ABSTRACT

Mechanisms for fault-tolerance in MPSoCs are mandatory to cope with faults during fabrication or product lifetime. For instance, permanent faults on the interconnect network can stall or crash applications even though the network has alternative fault-free paths to a given destination. This paper presents a novel fault-tolerant communication protocol that takes advantage of the NoC parallelism to provide alternative paths between any source-target pair of processors, even in the presence of multiple faults. The proposed approach determines new paths quickly, and the costs of extra silicon area and memory usage are small.

Keywords

NoC-based MPSoC; fault-tolerant communication protocol; fault-tolerant message passing protocol.

1. INTRODUCTION

Aggressive scaling of CMOS process technology allows the fabrication of highly integrated chips, enabling the design of multiprocessor system-on-chip (MPSoC) connected by network-on-chip (NoC), increasing the chip failure rate. This paper proposes a novel fault recovery communication protocol for MPSoCs. The method is built on top of a small specialized network used to search fault-free paths, and an MPI-like software protocol, implemented at the kernel layer, *hidden from the application layer*. The software protocol detects unresponsive processors via acknowledgment messages and automatically fires the path search network, which can quickly return a new path to the target processor. This approach has a *distributed nature*, thus, there is no single point of failure in the system, and provides *complete reachability.*

A reliable system must be able to mitigate, detect, locate faults, reconfigure itself, and recover the affected logic. Fault detection and location (i.e. fault diagnosis) are the first challenge in the quest for dependable NoC-based MPSoC. As surveyed in [1], the universe of online test approaches for NoC-based designed includes the use of error control coding schemes, BIST for NoC routers, and functional testing. The second challenge toward a dependable MPSoC would be the system reconfiguration, also surveyed in [1]. Basically, it receives the fault location and changes the system configuration, masking regions with permanent faults. The predominant fault reconfiguration approaches are the use of spare logic (spare links, routers, or entire PE), network topology reconfiguration, and use of adaptive routing algorithm [1]. As we move to a macro architectural view of dependability in MPSoCs, there are two main communication paradigms that must be taken into account: *shared memory* and *message passing*, surveyed in [2] and [3]. Fault tolerant message passing libraries were proposed in the context of fault tolerant distributed systems, usually applied for cluster of computers, like show in [4] and [5]. The use of fault tolerant message passing libraries specifically designed for MPSoCs and aiming fault reconfiguration are scarce, like show [6]. Methods for detect faults are surveyed in [7] and [8]. Router architecture is present in [11].

2. REFERENCE ARCHITECTURE

Each MPSoC PE (Processing Element) contains an IP connected to a NoC router. The IP has the following modules: (*i*) a 32-bit processor (MIPS-like architecture); (*ii*) a local memory; (*iii*) a DMA module, enabling parallelism between computation and communication; (*iv*) a network interface. The NoC adopts a 2D-mesh topology, with input buffering, credit-based flow control, and duplicated physical channels (i.e. each router has 10 ports). The distributed XY routing algorithm is adopted as the standard routing mode between PEs. The network also supports source routing such that it is possible to determine alternative paths to circumvent hot spots or faulty areas. At the software level, each processor executes a *microkernel*, responsible for communication among tasks, management services, and multi-task execution. Message passing is the communication method adopted in this work. Applications that adopt message passing are modeled as task graphs, with vertices being tasks and edges the messages between them.

3. FT COMMUNICATION PROTOCOL

The proposed method supports permanent faults only on the NoC, exploiting the redundant paths of the network. The faults can lead to payload error (detected by error detection codes) or packet-routing errors, detected by timeout approaches. This paper focuses on the second kind of fault effect because it is typically harder to recover than the first kind of fault. The proposed FT communication protocol is divided in three layers:

- ***Application Layer***: The task code responsible for calling the execution of *Send*() and *Receive*() primitives. The application layer was not modified.

- ***Kernel Layer***: Transfers the messages from the task memory space to the kernel memory space, and transmit messages from the NoC to the task memory space;

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

3.1 Kernel Layer

When part of the NoC is isolated by faults, eventually some packet will not be delivered. The kernel layer has been modified to detect these undelivered messages and to perform retransmission. These were the main kernel modifications:

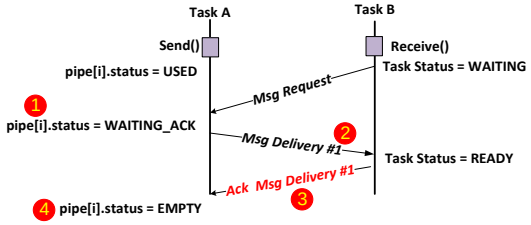
All data packets are locally stored in the *pipe* before being sent to the NoC. This enables the packet retransmission since the source PE temporally keeps a local copy of the packet;

For all delivered packets, an acknowledgment packet is transmitted from the target to the source PE;

Each packet generated by a single PE receives a unique sequence number.

Figure 1 details the proposed fault-tolerant protocol. The slot with the message being transmitted assumes the status *waiting*

acknowledgment (label 1 in ,Figure 1). When the message is received, its sequence number is verified. If it is the expected sequence number, the task can be scheduled to run, the message is consumed (2), and the acknowledgment packet with the sequence number is transmitted to the source PE (3). The last step of the protocol is to release the pipe slot, assigning to its position an *empty* state (4).



,Figure 1 – Communication protocol modified to cope with the FT communication protocol.

3.2 Hardware Layer

At the hardware layer two modules were modified: NoC and NI. The NoC received the path search module, called *seek module*, briefly presented in this section and detailed in [9][1]. This module performs three steps: (*i*) *seek* new path; (*ii*) *backtrack* the new path; (*iii*) *clear* the seek structures and *compute* the new path. The main feature of this routing method is the adoption of a small and dedicated network to discover the fault-free path. The new path search is executed only once, when the unresponsive task is detected, returning a new path to the source PE. This path must be stored for future packet transmissions, using source routing. In practice, each PE communicates with a limited amount of PEs, enabling to use small tables for path storage. Once the new path is determined, the application's protocol latency returns to its original performance.

3.3 FT to the Communication Protocol

Faults in the network may interrupt the communication protocol in four main situations: *(i)* fault in the message delivery (section 3.3.1); *(ii)* fault in message acknowledgment (section 3.3.2); *(iii)* fault in message request (section 3.3.3); *(iv)* fault during the packet receive (section 3.3.4).

3.3.1 Fault in the Message Delivery

This scenario, illustrated in Figure 2, shows the protocol diagram when there is a fault in the path from A to B.

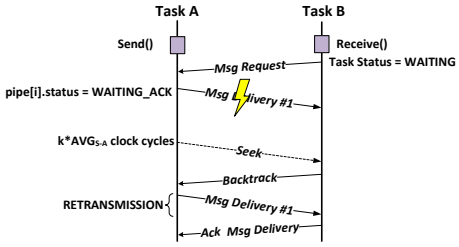


Figure 2 – Protocol diagram of a message delivery fault.

Task B requests the message to Task A. Task A sends the message delivery packet, and waits for the acknowledgement packet. An adaptive watchdog timer, detailed in Section 3.4, is used. The average time between each *send* and *acknowledgement* is computed, being defined as AVG_{S-A}. If the acknowledgement is not received after k*AVG_{S-A} (k is discussed in Section 3.4), the target router is declared unreachable. Each time the processor schedules task A, the scheduler verifies if it has messages to task B in the *pipe* stored for more than k*AVG_{S-A} clock cycles. If this condition is true, the fault-tolerant routing method is executed, and the faulty-free path stored in the NI. After computing the new path, the packet is retransmitted to task B, and the acknowledgement is received by task A.

3.3.2 Fault in Message Acknowledgment

In this scenario, illustrated in Figure 3, task B requests message from A, A sends the message to B, and B sends the acknowledgment back. However, the acknowledgment is not received due to a fault in the path from task B to task A. So, task B successfully received the message, but the problem is that task A cannot release the *pipe* slot having the consumed message.

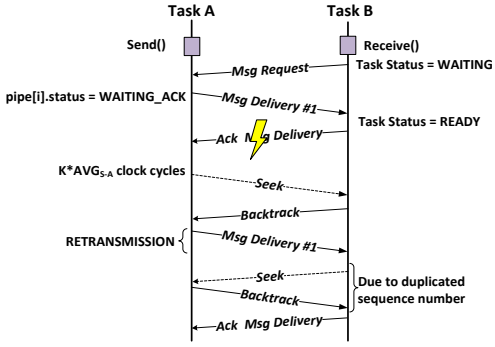


Figure 3 – Protocol diagram of ack message fault.

Therefore, from this fault, the adaptive watchdog timer will interrupt the PE holding task A (fault in the message delivery). As can be observed, the search for the new path would not be needed, since the path A→B is faulty-free, but it is impossible for task A to know why the acknowledgment was not received. This step corresponds to the "retransmission" label in Figure 3. Task B, in this case, receives the same packet with the same sequence number. This packet is therefore discarded by task B. Then PE holding task B starts the seek process to the PE holding task A, to find a faulty-free path for the acknowledgment packet. Once the new path to task A is received, the acknowledgment packet is transmitted.

3.3.3 Fault in Message Request

Figure 4 shows the protocol diagram for a message request that was not received Figure 2. The solution to detect a fault in this case is to adopt an adaptive watchdog timer at the task B side. The average time between requests is recorded in the variable AVG_{req}. Elapsed k*AVG_{req} clock cycles the seek process is executed, and the last message request is retransmitted using the new path.

[1] [9] describes and evaluates the seek module (i.e. it details Section 3.2) while the present paper presents the protocol on top of the seek module.

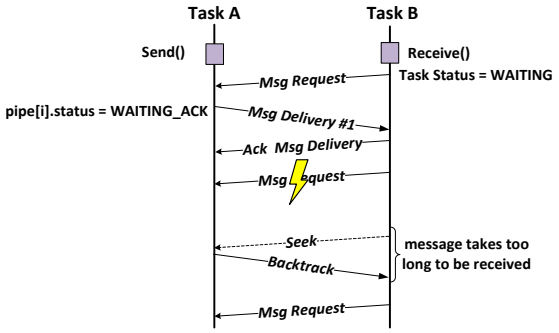


Figure 4 – Protocol diagram of message request fault.

3.3.4 Fault Receiving Packets

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

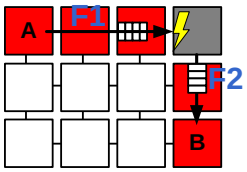


Figure 5 – Example of a faulty router in the middle of a message being delivered

The practical outcome is that all flits of flow F1 are virtually consumed by the faulty port, avoiding fault propagation. This ensures that any data sent to a faulty port is immediately dropped, and F1, in this case, disappears.

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

3.4 Adaptive Watchdog Timers

The adoption of adaptive watchdog timers at both sides of the communication has as main advantage to adapt the timers to the application profile. Figure 6 represents six protocol transactions and their respective protocol latencies, assuming a faulty and a fault-free transaction. These protocol latencies are used to calculate the AVG, the *average protocol latency*. The term AVG refers to both $AVG_{S\text{-}A}$ and $AVG_{req.}$ K is a constant defined at design time per application. Once the threshold of *k*AVG* is reached, the proposed method is fired to determine a new path and to retransmit the packet via this new route. Figure 6 also illustrates the main components of the protocol latency when a fault is detected (PL_f). It can be seen that most time is spent in *k**AVG. Next, the proposed method is fired, spending a small amount of time to find the alternative path (TFAP). Equation 1 summarizes the components of PL_f.

$$PL_f = k*AVG + TFAP + retransmission \quad (1)$$

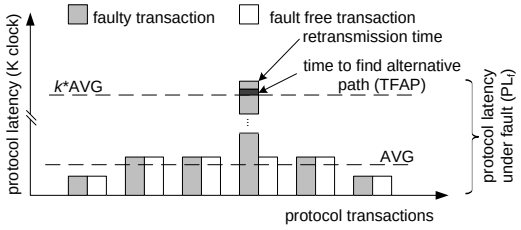


Figure 6 – Comparing faulty and fault-free protocol latency.

4. RESULTS

This section presents the experimental setup, detailing the evaluation flow and the applications used as case study. The first part (Section 4.2) presents the results obtained from a functional validation of the proposed approach, demonstrating its functionality. The last part (Section 4.3) evaluates the silicon area overhead and the memory used to implement the software (kernel) part.

4.1 Experimental Setup

4.1.1 Evaluation Flow

A small 3x3 network has 48 ports (excluding the ports in the chip's boundary) in total. The number of possible fault scenarios grows exponentially with the number of simultaneous faults. For instance, 1 fault requires C(48,1) = 48 scenarios and 2 faults requires C(48,2) = 1128 scenarios. For this reason we created an automatic and parallel fault analysis flow, divided into five main phases.The first phase, ***MPSoC Generation***, is responsible for generating and compiling the hardware and software of the MPSoC. The processor and local memory are described in cycle accurate SystemC and the other modules are described in RTL VHDL. The ***Simulation Scenario***, which consists of the compiled kernel, the compiled application, and the MPSoC hardware model. The ***fault scenario generation phase*** generates a database of faults scenarios to be evaluated. The ***fault simulation phase*** executes a fault simulation for each scenario in the database. A grid computing resource is used to distribute the simulation jobs in parallel among workstations. Each simulation generates logfiles with the results. These log files are parsed in the ***result analysis phase***, extracting the useful performance information and checking whether the application was able to execute with faults. This phase also generates regression reports, charts, and tables used to compare each fault scenario.

4.1.2 Evaluated Applications

Consists in two synthetic applications (called *basic* and *synthetic* applications), used mainly for validation purposes, and a third real application (with actual computation) that implements part of an *MPEG* encoder. These three applications were decomposed into communicating tasks which are illustrated in Figure 7.

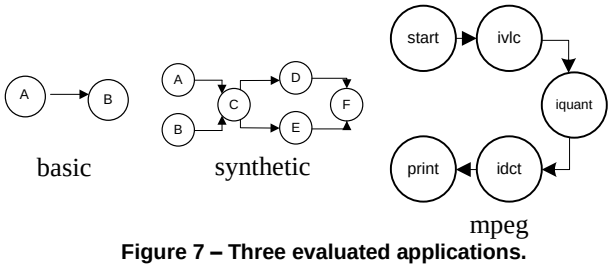


Figure 7 – Three evaluated applications.

4.2 Validating the Proposed Approach

These first set of experiments were designed to validate the proposed approach. Scenarios with 1 and 2 simultaneous faults were generated. Since the first goal is functional validation, we initially assume k = 12 for each application because this value is large enough such that false seek requests are not fired. Table 1 summarizes the validation process for single fault scenarios. The evaluated criteria are:

- ***scenarios***: number of simulated scenarios;
- ***scenarios(%)***: the percentage of possible scenarios. 100 means that all possible fault scenarios were simulated;
- ***affected-scenarios***: number of scenarios affected by faults. In this context, affected means that at least one task fires a seek request;
- ***faulty-scenarios:*** the number of fault scenarios which caused system stall;
- AET_n: the Application Execution Time (in ms) assuming a fault-free system;
- $AET_{max\text{-}f}$: the maximal Application Execution Time (in ms) obtained considering the scenarios affected by faults;
- ***time(%)***: the overhead in simulation time caused by the faults defined as $AET_{max\text{-}f}$ /AET_n.

Table 1 – Validation results with 1 fault.

	basic	synth	mpeg
scenarios	48	48	48
scenarios (%)	100	100	100
affected-scenarios	8	12	8
faulty-scenarios	0	0	0
AET_n (ms)	0.9061	3.0767	5.2302
$AET_{max\text{-}f}$ (ms)	1.5174	4.1840	8.8116
time(%)	67.46	35.99	68.47

Table 1 demonstrates that at least 8 fault injections affected the application execution (*affected-scenarios* row), however, the proposed approach was able to find an alternative path, enabling the application to finish (*faulty-scenarios* row) its execution. The maximal application execution time ($AET_{max\text{-}f}$ row) of these affected scenarios were 1.51, 4.18, 8.81 milliseconds for each application. Likewise, Table 2 details similar information, but assuming two faults per scenario. Thus, 17% of the scenarios were simulated for *basic*, *synthetic*, and *mpeg*.

Table 2 – Validation results with 2 faults.

	basic	synth	mpeg
scenarios	191	191	191
scenarios (%)	17	17	17
affected-scenarios	40	67	50
faulty-scenarios	2	3	3
AET_n (ms)	0.9061	3.0767	5.2302
$AET_{max\text{-}f}$ (ms)	1.5174	4.2700	8.8154
time(%)	67.46	38.79	68.55

The results in Table 2 show increased number of affected fault scenarios (*affected-scenarios* row) but, for most of them, the proposed approach was able to determine an alternative path. However, there were some system stalls (*faulty-scenarios* row). The reason is that tasks might be isolated by faults such that the router is unreachable.

4.3 Silicon Area and Memory Usage

Table 3 shows the area overhead comparing our approach to the baseline MPSoC [10]. There are two scenarios to be considered: the new path vector implemented in software, in the kernel space memory; or it can be implemented in hardware, in the Network Interface.

Table 3 – Area overhead (with and without hardware tables) compared to baseline MPSoC. Target device: xilinx xc5vlx330tff1738-2.

Module		MPSoC[10]	FT wo/ table	overh.	FT w/ table	overh.
NI	LUTs	225	308	37%	943	319%
	FFs	137	171	25%	420	207%
DMA	LUTs	157	166	6%	167	6%
	FFs	121	123	2%	123	2%
router	LUTs	1702	2326	37%	2345	38%
	FFs	450	690	53%	690	53%
PE	LUTs	4395	4999	14%	5604	28%
	FFs	1156	1440	25%	1720	49%

The memory usage overhead for the kernel with the proposed FT is 6.5 KB. The memory usage overhead in [8] is 8.1 KB.

5. CONCLUSION

This work presented a FT communication protocol for NoC-based MPSoCs. Both supporting hardware and software were fully integrated and validated on an existing MPSoC design described in RTL. The proposed method was evaluated with synthetic and real applications with permanent faults. The protocol automatically detects the unreachable tasks and launches the search for a faulty-free path to target PE. The overhead in silicon area and kernel's memory are acceptable.

6. REFERENCES

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

[2] F. Fu; *et al*. "MMPI: A Flexible and Efficient Multiprocessor Message Passing Interface for NoC-based MPSoC". In: SOCC 2010, pp. 359–362.

[3] Mahr, P.; *et al*. "SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips". In: International Conference on Reconfigurable Computing and FPGAs, 2008, pp. 187–192.

[4] R. Aulwes and D. Daniel. "Architecture of LA-MPI, a Network-Fault-Tolerant MPI". In: Parallel and Distributed Processing Symposium. 2004.

[5] R. Batchu; *et al*. "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware". Cluster Computing, vol. 7(4), pp. 303–315, 2004.

[6] Kariniemi, H. and Nurmi, J. "Fault-Tolerant Communication over Micronmesh NOC with Micron Message-Passing Protocol". In: Symposium on SoC, 2009, pp. 005–012.

[7] Zhu, X. and Qin, W. "Prototyping a Fault-Tolerant Multiprocessor SoC with Run-Time Fault Recovery". In: DAC 2006, pp. 53–56.

[8] Hebert, N.; *et al*. "Evaluation of a Distributed Fault Handler Method for MPSoC". In: ISCAS 2011, pp. 2329–2332.

[9] Wächter, E.; *et al*. "Topology-Agnostic Fault-Tolerant NoC Routing Method". In: DATE 2013. to appear.

[10] Carara, E.; Moraes, F. "Flow Oriented Routing for NOCS". In: SOCC 2010, pp. 367–370.

[11] Rodrigo, S.; *et al.* "Cost-Efficient On-Chip Routing Implementations for CMP and MPSoC Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30(4), pp. 534–547, 2011.