Runtime Fault Recovery Protocol for NoC-based MPSoCs

Eduardo Wächter*†, Augusto Erichsen†, Leonardo Juracy†, Alexandre Amory† and Fernando Moraes†

† FACIN – PUCRS – Av. Ipiranga 6681– Porto Alegre – RS – Brazil
* Santa Cruz do Sul University (UNISC) – Faculty of Informatics – Brazil
eduardo.wachter@acad.pucrs.br, {alexandre.amory, fernando.moraes}@pucrs.br

ABSTRACT

The design of reliable MPSoCs is 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. At the application layer, the method is seen as a typical MPI-like message passing protocol. At the lower layers, the method consists of a software kernel layer that monitors the regularity of message exchanges between pairs of tasks. If a message is not delivered in a certain time, the software fires a path finding mechanism implemented in hardware, which guarantees complete network reachability. 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.

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). However, the increase of integration level also increases the chip failure rate. The NoC, for instance, can be especially sensitive to faults since a single permanent fault on the NoC can stall the entire system.

On the other hand, NoC-based MPSoCs provide excellent opportunities for the design of reliable systems. For example, there are *redundant paths* between processors and it is possible to design an *integrated hardware-software approach*, combining the performance of hardware and flexibility of software.

To cope with unreliable NoCs, 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*, since a source processor uses only local information to monitor the path to the targets, thus, there is no single point of failure in the system. The proposed approach provides *complete reachability*, i.e. if there is only one fault-free path to the target, it will be found, even in the presence of *multiple faults*.

The rest of this paper is organized as follows. Section 2 reviews fault tolerant (FT) MPSoCs. Section 3 presents the required background to deploy the FT communication protocol. Section 4 presents a motivational example to demonstrate the need for FT communication. Section 5 details the proposal, and Section 6 evaluates its performance. Section 7 concludes this work.

2. RELATED WORKS

A reliable system must be able to mitigate, detect, locate faults, reconfigure itself, and recover the affected logic [1]. Fault *mitigation* approaches can reduce the appearance of faults. Those faults that still exist must be *detected* and *located*. Once the fault is located, the system (hardware and software) must be *reconfigured* in order to avoid the defective part. Finally, if there is corrupted data or task, it must be *recovered* to its last known good state. Thus, to achieve fault tolerance in MPSoCs a layered approach is required. Such layers may be defined as:

i. *path reconfiguration layer*, where the original faulty path is replaced by an alternative healthy path [2][3];

ii. *protocol recovery layer* [4][5][6][7][8], where the communication protocol retrieves data lost during a faulty data transfer;

iii. *system reconfiguration layer* [9][10], where faults in the processing element (PE) require support to task migration (move tasks from the defective PE to a healthy PE), online task mapping (decide the new task location in runtime), and saving the task's internal state (i.e. checkpoints and rollback).

Rodrigo et al. [2] 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 propose the replication of the packets in two output ports, ensuring complete reachability, but increasing network congestion. Wächter et al. [3] present a fault tolerant routing algorithm, which is topology independent.

Fu et al. [4] describe an MPI-based library where the task mapping is abstracted from the application. It also implements features such as broadcast-gather and message ordering. Mahr et al. [5] argue that most message passing libraries for MPSoCs are specific for a single network. They developed a network-independent library that, according to the target network topology, can be simplified or customized, removing the unnecessary parts and reducing the memory footprint.

Aulwes and Daniel [6] include reliability features on MPI such as checksum, message retransmission, and automatic message re-routing. Batchu et al. [7] test unresponsive processes by implementing self-checking threads that use heartbeat messages generated to monitor the MPI/FT progress. On top of this test approach, they built a recovery mechanism based on checkpoint.

The use of fault tolerant message passing libraries specifically designed for MPSoCs and aiming fault

reconfiguration are scarce. For instance, Kariniemi and Nurmi [8] present a similar fault-tolerant communication library targeting MPSoCs based on packet retransmission, where timers and CRCs are used to detect packet losses and bit errors caused by transient faults at the NoC links.

Zhu and Qin [9] propose a fault-tolerant MPSoC executing a single program, multiple data (SPMD) environment. In the SPMD approach, all PEs execute copies of the same data-parallel program. Then, if a tile fails, there is no instruction loss, because another tile may execute the same program. The proposed framework uses an MPI-like message-passing library. The authors evaluate a DSP application that has been replicated 4 times. This application was mapped in a 4x4 MPSoC, showing a speed up varying from 1 to 16 working PEs.

Hébert et al. [10] proposes a distributed fault-handling method for a NoC-based MPSoC. Faults at the PE are detected with heartbeat messages exchanged among PEs. Each PE has a watchdog timer capable of detecting missing or corrupted heartbeat messages. Besides, fault detection, the method also comprises faulty PE isolation and recovering via task remapping. As stated at the conclusion of [10], the approach does not handle faults at the interconnect network.

This review of prior work demonstrates that there are a large number of approaches, but addressing only one layer toward the design a dependable MPSoCs. The *contribution of this paper* lies on the second layer, *protocol recovery*, coupled to the first layer, *path reconfiguration*, proposed in [3]. The resulting MPSoC is a framework enabling to explore *system reconfiguration* techniques.

3. REFERENCE ARCHITECTURE

Each PE contains an IP connected to a NoC router, as illustrated in Figure 1. 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 (NI). 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. Figure 1 also shows the location of the test wrappers cells.

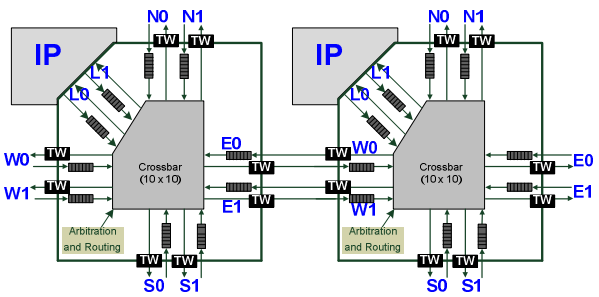

Figure 1 – PE architecture, detailing the internal router structure (TW: test wrappers cells).

At the software level, each processor executes a small operating system, name *microkernel*, responsible for communication among tasks, management services, and multi-task execution. Message passing is the communication method adopted in this work. Tasks communicate using two MPI-like primitives: a non-blocking *Send()* and blocking *Receive()*. To implement an non-blocking *Send()*, a dedicated memory space in the microkernel, named *pipe*, stores each message written by tasks.

Figure 2 illustrates a communication between two tasks, A and B, mapped in different PEs. When task A executes a *Send()* (1), the message is stored in the *pipe*, and computation continues (2). This characterizes a non-blocking writing. When task B executes a *Receive()* (3) the microkernel sends a request message through the NoC (4) and the task enters in wait state (5), characterizing a blocking read. Task A receives the message request (6), and sends the message through the NoC, freeing the space in the *pipe* (6). When the message arrives in the task B (7), the microkernel stores the message in the task B memory space, and returns to execution status (8).

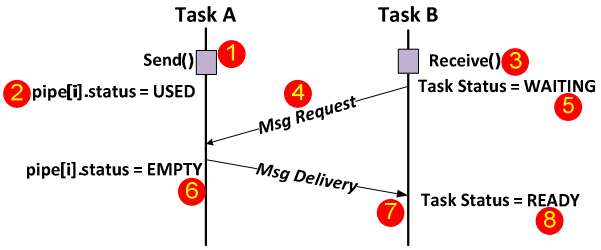

Figure 2 – Communication protocol adopted in the reference MPSoC.

3.1 Isolation Method

Complex systems such as SoCs must have wrapped cores such that each core can be tested individually. The test wrapper isolates the Core Under Test from the rest of the system and we propose to use this existing feature also for fault tolerance purposes.

Each router output port has a set of test wrapper cells (TW), enabling to isolate a faulty port, due to faults in the output port, physical link or input port, as detailed in port Figure 3(a). If a fault affects the router's control logic, then the entire router can be isolated by disabling all ports. Such approach allows using the router and the PE even if some part of the router is faulty. Figure 3 (b) represents how the physical fault is represented graphically along this paper.

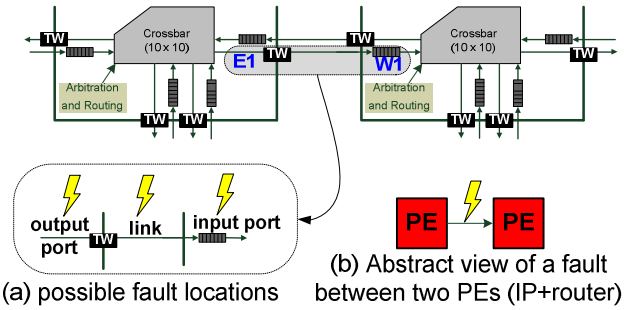

Figure 3 – PE architecture, detailing the internal router structure (TW: test wrappers).

The isolation method requires minimal extra hardware and can be adapted to other router designs with different number of ports. Moreover, it enables to gracefully reduce the network performance as the number of faults increases since it is possible to shut off one router port individually. Finally,

the *fault detection method is decoupled* from the rest of the fault recovery approach. It just has to write in the *faulty_port* register when it detects a defective router port.

4. MOTIVATIONAL EXAMPLE

Assume in Figure 4 a fault in the path between tasks A and B. Using a conventional message passing library with no FT mechanism, a *Receive()* call can wait indefinitely if an unreliable communication layer loses the message. In addition, a packet hit by a fault might be misrouted or occupy buffers indefinitely and a faulty router might generate wrong signals to its neighbors (byzantine fault). Therefore, to avoid system crashes caused by hardware faults, the system must *locate*, *avoid*, and *isolate* the faulty region. For instance, if the fault is permanent and the network uses only deterministic routing algorithm, then the packet retransmission would use the same faulty path, generating a new pending packet. Thus, the system must have the ability to *locate the faulty router* and support *adaptivity*, delivering packets using an alternative path and *avoiding* the faulty region.

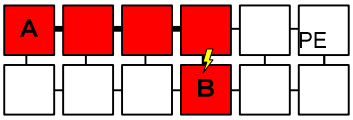

Figure 4 – Motivational example to justify the adoption of a fault-tolerant communication protocol. Red PEs represent the path taken by packets assuming an XY routing algorithm.

5. SEEK NETWORK

At startup, the network is assumed fault free and packets are normally sent from the source PE to the target PE. Once the packet reaches its destination, the target PE must deliver an acknowledgment packet to the source PE. If the *ack* packet is not delivered in a given time, the source PE assumes the original path is faulty, and triggers the proposed path search method. This method, executed at runtime, comprises three steps, as illustrated in Figure 5: (*i*) seek new path; (*ii*) backtrack the new path; (*iii*) clear the seek structures and compute the new path. At the end of the described process, the source PE receives a valid path to the target PE. This path is stored in the PE memory and the following packet transfers (from this source to the target) use this path. Thus, the proposed method is executed only once for each missed *ack* packet, without requiring the whole seek process again for each packet transfer.

In the first step of the method, ***seek step***, *if* the source PE identifies that it is not able to transfer packets to a target PE, this PE generates a *seek request*, asking for a fault-free path to the target PE. The router stores this request into an internal table containing the fields **S/T/P/#**, meaning source router, target router, incoming port, and hop counter. Each router checks if it is the target router. If not, it increments the hop counter and broadcast the *seek request* to its neighbors (except for the port which originated the request), via a separate network optimized for broadcast. The next routers store the *seek request* in their own internal tables, checking if they are the target router and repeat the process until the target router is reached or a timeout mechanism at the source PE is fired. In this case, the source PE broadcasts a clear command to free the seek tables and it tries another seek a random time latter. The target router is declared unreachable after a given number of unsuccessful seek requests.

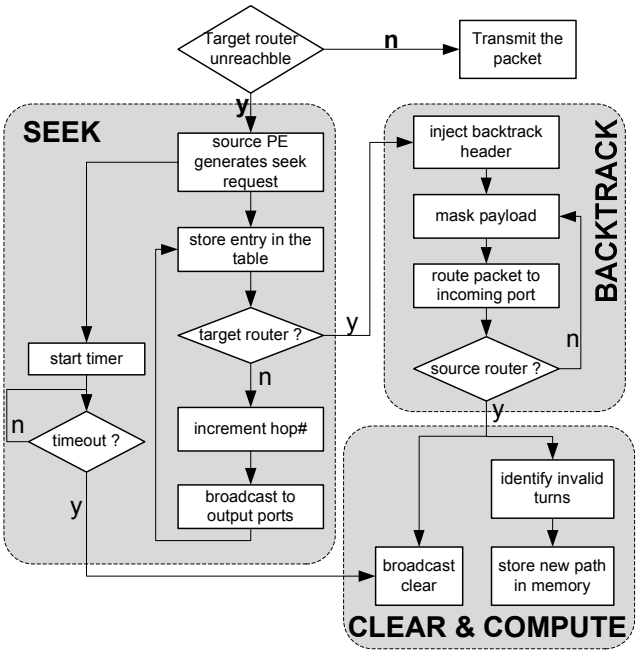

Figure 5 – Flowchart of the routing method.

Figure 6 illustrates a scenario with **four** faulty routers, one source, and one target router. Initially, the source PE requests a seek to its router, creating an entry in its local table with value S/T/L/0, meaning that the source request came from the local port (L) and it is hop 0 (Figure 6(a)). Since this is not the target router, it broadcasts the seek request to the north port. The second router (Figure 6(b)) stores the entry S/T/S/1 in its local table, meaning it received the request from the south port and it is the hop number 1. The broadcast continues until it reaches the target router (Figure 6(d)) via the east port with 8 hops.

Figure 6 –Seek steps of the method in a 4x4 mesh, with four faulty routers.

The second step, ***backtrack step*** is executed only once, and starts when the target router is reached by the first received seek. All subsequent seeks are discarded. The target router injects a *backtrack packet* into the NoC, where its payload correspond to the ports selected by each router to reach the target router. The *backtrack packet* uses source routing, following the reverse path followed by the seek step. The

backtrack ends when the source router is reached and the backtrack packet is sent to the source embedded processor to check the validity of the path.

The backtrack packet may not be received in two cases: the target router is unreachable, or the number of maximal simultaneous seeks is exceeded. In the second case, the source router can initiate a new seek after a random time. If all the seek requests failed to receive backtrack, then the target router is declared unreachable.

The third step is named ***compute path and clear seek***. The path followed by the backtrack packet might create a deadlock. For this reason, the source embedded processor checks the path, for instance [N N E E E S S W E], for invalid turns. When there is an invalid turn, it forces the packet to change the physical channel (or virtual channel), breaking the dependency cycle. For instance, the path presented above has a turn from south to west. This path should be changed from channel 0 to 1 because it is a prohibited turn in the west-first routing algorithm, but allowed by east-first one. Thus, the corrected path to the example target router must be [N0 N0 W0 W0 W0 S0 S0 W1 E1]. Once this path is computed, it is stored in the PE main memory for future packet transmissions.
In parallel to the path computation, the source PE requests to clear the entries in the seek table labeled with the source router address. This clear command is broadcasted to the entire NoC, similarly to the seek process.

6. 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. Therefore, the software designed for the original platform, may be used with the new fault-tolerant protocol without modifications;
- ***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 NI and the NoC routers.

6.1 Kernel Layer

When part of the NoC is isolated, 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 7 details the proposed fault-tolerant protocol. Different from the original communication protocol presented in Figure 2, the source PE does not release the pipe when a message is requested. The slot with the message being transmitted assumes the status waiting acknowledgment (label 1 in Figure 7). 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).

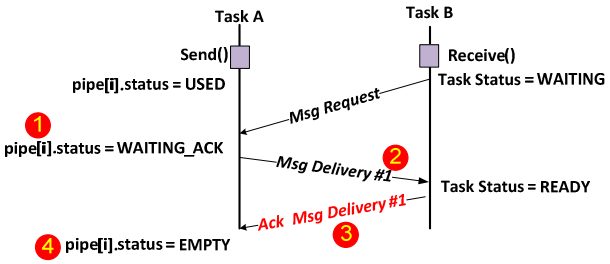

Figure 7 – Communication protocol modified to cope with the FT communication protocol.

6.2 Hardware Layer

At the hardware layer two modules were modified: NoC and NI. The NoC received the path search module, presented in section 5 and detailed in [3]. The new path search is executed only once, when the unresponsive task is detected. The search returns 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. Thus, the *path storage table* can be implemented either in hardware or in software.

The advantages of this method are: (*i*) the *network latency* for packets sent with XY or source routing is the same, reducing the performance penalties on faulty NoCs; (*ii*) the number of entries in the path storage table is not a function of the NoC size, characterizing a *scalable* method; (*iii*) the path finding process is *fast* to define a new faulty-free path (discussed in the results). Once the new path is determined, the application's protocol latency returns to its original performance.

6.3 Adding 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 6.3.1); *(ii)* fault in message acknowledgment (section 6.3.2); *(iii)* fault in message request (section 6.3.3); *(iv)* fault during the packet receive (section 6.3.4).

6.3.1 Fault in the Message Delivery

This scenario, illustrated in Figure 8, shows the protocol diagram when there is a fault in the path from A to B. 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 6.4, 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 6.4), 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.

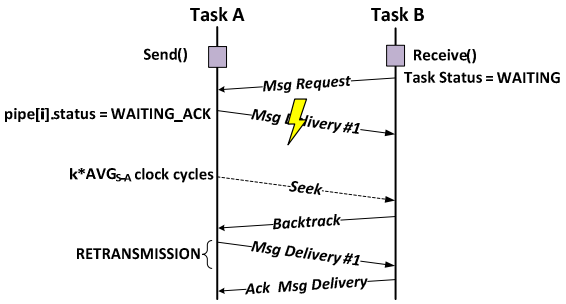

Figure 8 – Protocol diagram of a message delivery fault.

6.3.2 *Fault in Message Acknowledgment*

In this scenario, illustrated in Figure 9, task B requests message from A, A sends the message to B, and B sends the acknowledgment back to A. However, the acknowledgment is not received due to a fault in the path from task B to task A. In this scenario, task B successfully received the message, but the problem is that task A cannot release the *pipe* slot having the consumed message, since it assumes that the message was not received.

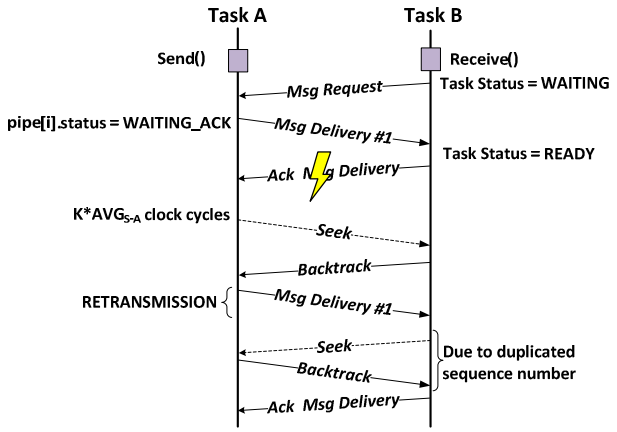

Figure 9 – Protocol diagram of ack message fault.

Therefore, from this fault, the adaptive watchdog timer will interrupt the PE holding task A, inducing the process presented in the previous session (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 9.

Task B, in this case, receives the same packet with the same sequence number. This packet is therefore discarded by task B, signalizing an error in the acknowledgment path due a wrong sequence number. 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.

6.3.3 *Fault in Message Request*

Figure 10 shows the protocol diagram for a message request that was not received (scenario similar to Figure 8). 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.

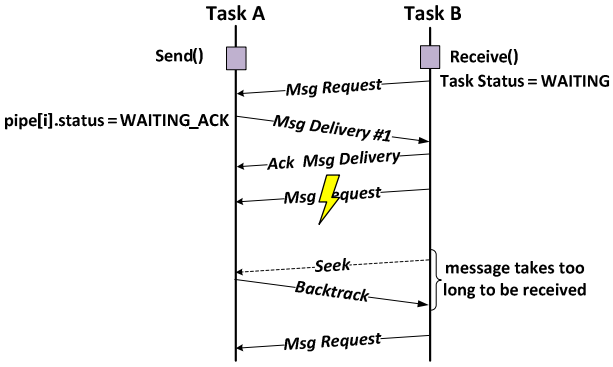

Figure 10 – Protocol diagram of message request fault.

6.3.4 *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 as illustrated in Figure 11. 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.

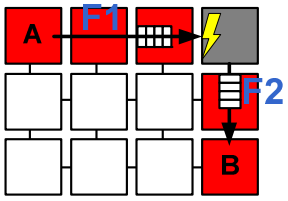

Figure 11 – Example of a faulty router in the middle of a message being delivered.

In fault mode, the wrapper logic around the faulty port signalizes that it can receive any flit transmitted to it. 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.

6.4 Adaptive Watchdog Timers

A fixed amount of time to start the *seek* process is not suitable for MPSoCs with applications inserted at runtime. A communication intensive application would stall for a long period if the fault timer threshold is too high. On the other side, a computation intensive application would generate false fault alerts, since these applications can take a long period between communications. 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 12 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.

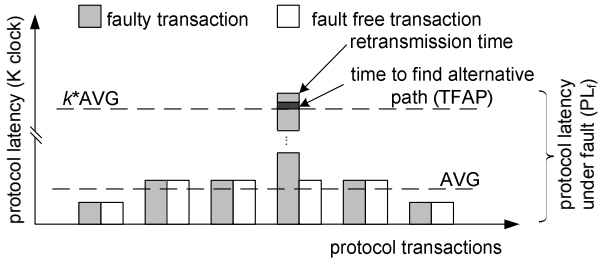

Figure 12 – Comparing faulty and fault-free protocol latency.

Figure 12 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). Finally, this new path is used for the packet retransmission. In addition, note in the last two protocol transactions that the protocol latency returns to normal. It happens because the path calculation occurs only when the fault is detected and the new path is used in all subsequent packet transmissions. Equation 1 summarizes the components of PL_f.

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

7. RESULTS

By using the isolation method (Section 3.1), it is possible to disable any individual router port of the NoC. The router has 10 ports, but the local port is not used for fault injection. For this reason, faults can be injected in 8 router ports. 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 48 scenarios combinations (C(48,1)=48) and 2 faults requires C(48,2) = 1128 scenarios. One can easily see that the number of fault simulations grows very fast and this evaluation cannot be executed manually. For this reason we created an automatic fault analysis flow, which include the MPSoC generation, the creation of a simulation scenarios, simulation of all scenarios and performance evaluation.

Three applications are used as benchmarks: two synthetic applications (*basic* and *synthetic*), used for validation purposes; and a real application (with actual computation) that implements part of an *MPEG* encoder. These three applications were decomposed into communicating tasks, which are illustrated in the task graphs of Figure 13. This figure also shows the task mapping into the MPSoC.

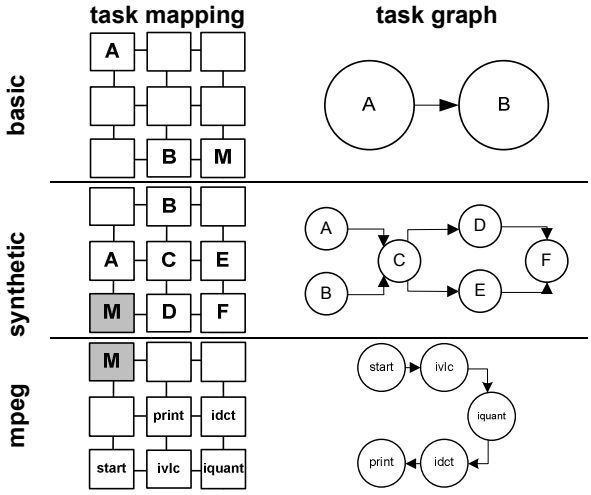

Figure 13 – Three evaluated applications, their task mappings and task graphs (M: manager processor).

7.1 Validating 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(%)***: 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:*** number of fault scenarios which caused system stall;
- AET_n: Application Execution Time (in *ms*) assuming a fault-free system;
- $AET_{max\text{-}f}$: maximal Application Execution Time (in ms) obtained considering the scenarios affected by faults;
- ***time(%)***: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

The results in Table 1 demonstrate that at least 8 fault injections affected the application execution (*affected-scenarios* row), and the proposed approach was able to find

an alternative path enabling the application to finish its execution (*faulty-scenarios* row). The maximal application execution time (AET_{max-f} row) of these affected scenarios were 1.51, 4.18, 8.81 milliseconds for each application. This corresponds to an overhead of 67%, 36%, and 68% (*time* row) compared to the normal execution time (AET_n row).

Likewise, Table 2 details similar information, but assuming two faults per scenario. In this case, it is not possible to simulate all possible fault scenarios since it would take several days of CPU time. 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-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, as illustrated in Figure 14. In this example, task A cannot send packets to task B because both outgoing ports (east and south) are faulty. Table 2 also shows that multiple simultaneous faults do not necessarily increase the AET_{max-f} since the seek request associated to each fault is executed in parallel.

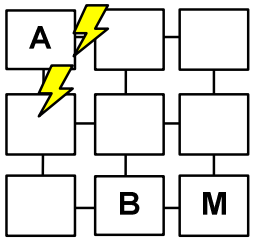

Figure 14 – Task isolation due to faults at east and south ports.

7.2 Faulty Application Execution Time

The TFAP (time to find an alternative path) consists of the time to perform the *seek*, *backtrack*, and *path computation* steps, as presented in Section 6.2. As observed in [3], the TFAP grows linearly with the number of hops between the source to the target task. Each hop of the *seek* and the *backtrack* steps takes about 15 clock cycles because they are executed in hardware. The *path computation* step takes about 200 clock cycles per hop, and it is executed in software (kernel layer). For instance, the TFAP for a path of 10 hops is about 2,300 clock cycles. The clock period in these experiments is 10 ns, thus, 2,300 clock cycles correspond to 23 *us* while the AET (application execution time) is in the order of *ms*.

The other relevant parameter for the AET is the time to fire the seek process, determined by *k**AVG. As discussed in Section 6.4, the AVG, the *average protocol latency*, is application-dependent. Figure 15 shows the *mpeg* AET considering the faulty-free scenario (label 0), and 2 single-fault scenarios (labels 1 and 2), for k = {2,4,8,12}. It is expected an increase in the ATE with the increase of K. In scenario 2 with $k = 2$ there is a slight increase in AET caused by false seeks induced by the low *k* value.

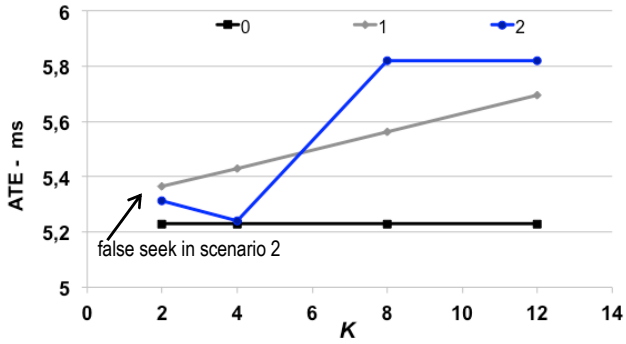

Figure 15 – Application execution time (AET) for the MPEG application, varying the K parameter (fault-free AET: 5.23 ms).

This experiment shows that the variation of AET under fault situations is small under typical application scenarios such as computation time >> communication time. Applications with more AVG variability requires larger *k*, thus they might suffer larger AET variability under fault conditions.

The proposed method can only affect the AET under normal condition (fault-free) if *k* is too small for the application. A simple communication application profile can help to indicate the appropriate *k*.

7.3 Silicon Area and Memory Usage

Table 3 shows the area overhead comparing our approach to the baseline MPSoC [11]. There are two scenarios to be considered: the new path vector implemented in software; or in hardware, in the NI. In the *software tables* approach, for each faulty target PE, the kernel stores the fault-free path in memory. Each time the PE sends a message to this target, the stored path is included into the packet header with source routing algorithm. In *hardware tables*, instead of storing in kernel memory, the new path is stored in a dedicated memory in the NI. Whenever the PE sends a packet to this target, the NI queries the table for an existing path, if it exists, the packet is sent with the stored path, if not, the packet is sent using distributed routing. The hardware table approach is implemented with 14 slots at the NI table, where each slot can store paths of up to 18 hops. Both slot and path length can be parameterizable according to the MPSoC size.

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

Module		MPSoC[11]	FT wo/ table	overhead	FT w/ table	overhead
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 hardware table approach has PE overhead of 28%/49% for LUTs and FFs, respectively. This is an acceptable cost,

since it ensures the maintainability of the application execution in the presence of faults. Without the tables, the area overhead is 14%/25% for LUTs and FFs, respectively. The memory usage overhead for the kernel with the proposed FT is 6.5 KB. The memory usage overhead in [10] is 8.1 KB.

8. 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 while permanent faults were injected into the NoC at runtime. The protocol automatically detects the unreachable tasks and launches the search for a faulty-free path to target PE. This way, the method enables applications to continue their execution, as long as there is at least a single functional path to the target PE. The entire process of finding alternative paths takes typically less than 2000 clock cycles or 20us. However, the time to fire the proposed approach might be longer, depending on application's communication features. The FT communication protocol, i.e., the addition of the acknowledgment message, caused less than 1% on the total execution time of computationally intensive applications, as the MPEG benchmark. The overhead in silicon area and kernel's memory are acceptable.

ACKNOWLEDGMENTS

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

[2] 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.

[3] Wachter, E.; Einsfeldt, A.; Amory, A.; Moraes, F. Topology-Agnostic Fault-Tolerant NoC Routing Method. In: DATE, 2013, pp. 1595-1600.

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

[5] 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.

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

[7] 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.

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

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

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

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