Analog Integrated Circuits and Signal Processing (2021) 106:85–98
https://doi.org/10.1007/s10470-020-01631-y

System management recovery in NoC-based many-core systems

Vinicius Fochi[1] · Luciano L. Caimi[2] · Marcelo H. da Silva[1] · Fernando Gehm Moraes[1]

Received: 31 July 2019 / Revised: 3 January 2020 / Accepted: 6 March 2020 / Published online: 12 March 2020

Abstract

The technology nodes reduction enabled the emergence of NoC-based many-cores with dozens to hundreds of processing elements (PEs). Despite the processing power offered by a large number of processors and communication flexibility due to the adoption of NoCs, it is necessary to manage the many-core resources to ensure scalability. The execution of the management tasks requires a PE reserved exclusively to execute such actions. These processors are named managers PE–MPE. A centralized approach would induce a significant load to the MPE in large-scale systems, and a permanent fault in the MPE would compromise the entire system. The adoption of a distributed approach, organization adopted in this work, with MPEs hierarchically organized, reduces the management load, and a fault in an MPE would compromise only the PEs managed by the faulty MPE. The literature presents several fault-tolerant proposals targeting the NoC or the processors. However, there is a significant gap related to fault-tolerant methods at the system level, i.e., related to fault-tolerant techniques regarding the MPEs. The goal of this paper is to present a recovery method when an MPE became faulty, and propose a protocol to migrate the management software safely to a new PE. The method adopts task migration to release a processor if there is no processor to receive the kernel that was executing in a faulty processor. The proposal is transparent to the applications running in the many-core, with an overhead in the execution time varying between 1.5 and 1.65 ms during the management and task migration.

Keywords Many-core · NoC · System management · Fault-recovery protocol · Task migration · Fault-tolerance

1 Introduction

Modern NoC-based many-core Systems-on-Chip (MCSoCs) enable embedded systems with dozens of processors. Large MCSoCs demand one or more dedicated Processing Element (PE) for management actions, as task mapping, task migration, power management (DVFS), QoS management, self-awareness adaptation, system monitoring (energy, temperature, deadlines). The management of a MCSoC may be centralized or hierarchically organized in clusters (Definition 1) [6, 9].

Definition 1 *Cluster*—an MCSoC virtual region, with a set of PEs and one manager MPE. The cluster size is a design-time parameter, but a cluster can borrow resources from other clusters at runtime when all clusters resources are busy—*reclustering* process [9].

With the ever-increasing reduction in the devices' geometry, transistors, vias, and wires degrade faster over time, causing transient and permanent faults to occur earlier, thereby decreasing the lifetime of integrated circuits [17]. For these reasons, reliability become a critical design

Author Vinicius Fochi is financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. Author Fernando Gehm Moraes is supported by FAPERGS (17/2551-196-1) and CNPq (302531/2016-5), Brazilian funding agencies.

✉ Fernando Gehm Moraes
fernando.moraes@pucrs.br

Vinicius Fochi
vinicius.fochi@acad.pucrs.br

Luciano L. Caimi
lcaimi@uffs.edu.br

Marcelo H. da Silva
marcelo.holgado@acad.pucrs.br

[1] School of Technology, PUCRS, Av. Ipiranga 6681, Porto Alegre 90619-900, Brazil

[2] UFFS, Av. Fernando Machado 108E, Chapecó 89802-112, Brazil

issue in MCSoCs [15]. Classical fault-tolerant approaches, as Triple Module Redundancy (TMR) or spare components [21], do not comply with today's requirements of silicon area and power dissipation. By construction, an MCSoC contains a set of replicated structures (the PEs), where a healthy component can replace the faulty component functions, with minimal performance reduction.

How to deal with a fault on MPEs opens a set of new challenges and opportunities in the field of many-core systems research. In a centralized approach, if the MPE presents a permanent fault, the entire system halts. In the hierarchical organization when an MPE became unresponsive, only a many-core region is affected.

Thus, MCSoCs need management mechanisms to perform different control tasks at the *system level*, and the technology evolution from one side enables to increase the number of PEs and from the other side accelerates the emergence of faults. The literature presents different approaches at the *system level*: power management (e.g., DVFS), performance and QoS management, resource management, and security [1, 7, 12, 25]. A rich literature with methods to test PEs is also available, with approaches adopted at different levels (hardware or software) or modules (Network-on-Chip (NoC), processors, memory). However, there is a significant gap related to fault-tolerant methods at the system level, i.e., related to the processors with the function to manage the system. Therefore, the MCSoC management requires monitoring and actuation policies to recover the system when one of these processors presents a permanent fault.

The *goal* of this paper is to present a recovery method when an MPE becomes faulty and proposes a protocol to migrate the management software safely to a new PE. The method does not require redundant structures, as TMR or software replicas. This work uses task migration and heuristics to select the new MPE position. The fault model assumes permanent faults on processors, and the NoC and memory have fault-tolerant mechanisms.

The paper is organized as follows. Section 2 presents related work. Section 3 details the system architecture and the fault model. Section 4 overviews the recovery method. Section 5 presents the actions executed by the recovery protocol. Section 6 details the recovery protocol steps. Section 7 presents the results, and Sect. 8 concludes this paper.

2 Related work

This sections reviews and discusses system management and fault-tolerance related work, summarized in Table 1. The 1st column contains the reference. The 2nd column shows the constraints applied to the system, which is management or fault-tolerance. The 3rd column presents the architecture (homogeneous, heterogeneous or only NoC) and the core counting given by the number of processor elements or the NoC size. The 4th column is the method or technique main goals under the constraint (2nd column). The 5th column lists the techniques used to control the system according to the Authors definitions. The last column presents the experimental setup used by the Authors and the abstraction level of the system modeling which the results are produced according to the following standard: (1) cycle-accurate simulation, only the execution time result is exact, and the others are estimated; (2) FPGA prototyping, cycle-accurate simulation for FPGA devices.

The literature presents distinct management and fault-tolerant approaches for MCSoCs that can be applied to the PE modules. According to Table 1, several techniques are used to manage the system with different goals, as power, resources, and performance. Fault-tolerance may be applied at different levels, as routing-algorithm, link level, processor level, system level. However, solutions that encompass a fault tolerance focused on the system management are scarce. Table 1 presents two works with fault-tolerance at the management cores [11, 26].

Domingues et al. [11], propose a system management technique targeting only the communication between PEs. The Authors propose a lightweight fault recovery mechanism for brokers (their MPEs) of a publish-subscribe middleware. The proposed approach uses the existing brokers to backup sensitive data of its neighbor brokers, which provides high availability to the system because when a fault is detected in a broker's processor, its neighbor broker promptly assumes the responsibility of managing the applications of the faulty broker. This method differs from our proposal since it targets only the communication management and not resources' management.

Tsoutsouras et al. [26] is the work most similar with our proposal. Tsoutsouras et al. [26] present a run-time resource management framework which can dynamically adapt the system to permanent faults in a self-organized, workload-aware manner. They proposed a self-organization that allows resource management agents to recovery from a fault electing a new agent to replace the faulty management agent, while workload awareness optimizes the election according to the status of each core. The work is hierarchically organized in: (1) controller cores: responsible for monitoring the system status sending this data to the set of the PEs; (2) manager core: responsible for managing one application; (3) worker cores: execute the applications' tasks. The cluster area is monitored by a controller core defined at system startup, but cannot change at runtime. To execute the recovery method in [26] it is necessary a communication protocol to update and leave the system in a safe state.

Table 1 Summary of the the state-of-the art

References	Design constraint	Architecture # of cores	Design goals	Techniques	Modelling (Exp. setup tools)
[20]	Resource management	Heterogeneous, 16 SPARC LEON3	Performance	Dynamic load distribution, adaptive shared resources	FPGA Simulation (Xilinx Virtex-5)
[18]	Power management	Homogeneous, 3×3 to 12×12	Scalability and energy efficiency	DVFS, clock gating, mapping, migration	Cycle-accurate simulation, and low-level analysis (Cadence tools)
[19]	FT on processor cores	Homogeneous 2×2	Task remapping	Task migration	FPGA simulation, Design Space Exploration (DSE)
[10]	FT on routing algorithm	Only NoC, 8×8	Adaptive routing algorithm, balance traffic load	Path discovery	Cycle-accurate simulation
[16]	FT on processor cores	Homogeneneous, 2–32 processing cores	CPU faulty detect	Software-based self-test routines	Xilinx ISEs WebPACK simulation
[2]	FT on links	Only NoC, 2×2 to 8×8	On-line test mechanism that detects stuck-at faults	Packet test	Xilinx 10.1 simulation
[3]	Management and FT on processing cores	Homogeneous, 6–12 processing cores	Performance, detected errors	Application's replicas	Cycle-accurate simulation + ReSP simulation environment
[11]	FT in Management Cores	Homogeneous 2–36	Fault detection and manager recovery	Migration of data contents and cluster reconfiguration	OVPSIM, instruction-accurate simulation
[26]	FT in Management Cores	Homogeneous 6–24 PE	Fault detection and manager recovery	Replacement core	Intel single-chip cloud computer (SCC)
[24]	FT in Processor cores	Homogeneous 4×4 to 10×10	Fault tolerant on critical applications	Task replicas	ORION 3.0 simulator
[23]	FT in data NoC routers	Homogeneous 2×2 to 8×8	Fault detection and routing reconfiguration	Deadlock-free routing computation	Cycle-accurate simulation
This work	FT in management cores	Homogeneous, parameterizable size	System management Recovery	Kernel manager mapping, task migration	Cycle-accurate simulation

FT Fault-tolerance

This paragraph describes the main differences from [26] to our work. Our proposal adopts one MPE per cluster, able to monitor several applications simultaneously, making the management infrastructure simpler. The management structure in [26] is complex, implying a penalty for system recovery in the order of seconds, against some milliseconds in our approach. The selection of the core to receive the faulty MPE is simpler in our method (Sect. 4), while [26] adopts a consensus agreement algorithm.

Most works on Table 1 present fault-tolerant methods focusing on the applications' execution, using methods well established in distributed systems. Fault-tolerance at the system level is a gap observed in the literature. The present work fulfills this gap, by proposing a runtime method to migrate the management functions assigned to a given MPE to a healthy PE.

3 System architecture and fault model

Figure 1 presents the reference many-core platform. It is an adaptation of the public-available HeMPS many-core [8]. The architecture contains a set of PEs interconnected by a *data NoC* and a *control NoC* [27]. The PE's hardware is the same, being the role assigned to the PEs made by software:

- *MPEs* PEs with management functions. The system contains Cluster Managers (CMs), responsible for managing a given cluster, and one CM with an interface with the external environment to receive new applications named Virtual Global Manager Processor (VGM);
- *Slave PEs (SPs)* execute applications' tasks, with an operating system supporting multitasking and message exchanging.

Each PE contains one processor (CPU), a Direct Memory Network Interface (DMNI, combining the functions of a Network Interface and a DMA module) [22], a dual-port private memory, the data and control routers, and a wrapper control module (WC) responsible for isolating the CPU in the presence of a fault using wrappers—*W*.

Two similar descriptions model the platform: (1) synthesizable VHDL, for characterization purposes; (2) RTL SystemC, with clock-cycle accuracy, enabling the simulation of systems with dozens of PEs.

The *data NoC* main features includes: (1) 2-D mesh topology; (2) 8-flit buffer depth input buffering; (3) wormhole packet-switching; (4) support for deterministic XY and source routing; (5) credit-based flow control; (6) duplicated physical channels (two 16-bit channels per link), enabling full adaptive routing.

The *control NoC* [27] transfers the control messages. The current work uses the *control NoC* to transmit messages with the following purposes:

- Notify the status of the MPEs;
- Freeze application(s) managed by a given MPE;
- Notify an SP that it will become a new MPE;
- Notify a DMNI module to transfer the memory contents of an SP to a new system address;
- Notity the SPs about a new MPE address;
- Unfreeze application(s) after the MPE migration.

An important architectural feature is that the memory is accessible by the data NoC, even if the processor has a permanent fault. The control NoC configures the DMNI module to transfer the memory contents to another PE. This feature of moving the memory contents when the processor has a permanent fault is commonly adopted in fault-tolerant approaches [19].

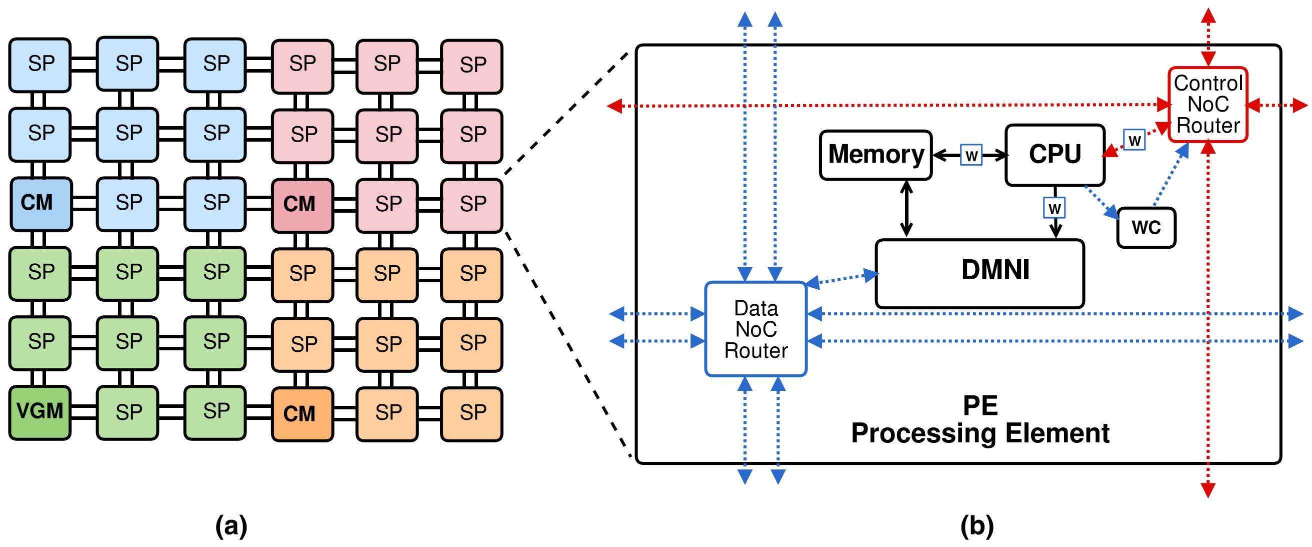

Fig. 1 **a** 6×6 MCSoC instance, with four 3×3 clusters. **b** Internal PE structure

The method herein presented may be applied to homogeneous or heterogeneous MCSoCs. The proposed method requires the following architectural features: (1) a set of PEs with the same architecture; (2) at least two disjoint NoCs, one for application data and one for management purposes [29]; (3) a memory module that can be read/write directly by the network interface [19].

The paper focus is *not* the fault detection, but the protocol for fault recovery. This work assumes:

- *PE healthy modules* memory, DMNI, data and control NoCs, wrapper control module. A usual method to protect the memory is the usage of ECC (Error Correction Codes). The DMNI is a small hardware module, with two state machines and a buffer. This module may be protected by hardware replication and adoption of ECC in the buffer. Besides the NoCs be considered healthy, it is possible to detect transient faults [14], and according to the transient faults severity, trigger the proposed protocol.
- *PE faulty module* The detection of a permanent fault in an MPE fires the proposed recovery method. The fault notification holds the faulty MPE and signalizes to the control NoC to send the fault notification. The control NoC [27] has an average latency of 14 clock cycles per hop, resulting in a minimal latency to deliver the fault message to the processor responsible for executing the recovery method.
- *Fault detection mechanisms* A rich literature with methods to test the modules of the processing elements is available, with approaches adopted at different levels or modules. Fault detection at the system level [4, 19], fault detection at the router level [13, 30, 31], fault detection at the processor level [5, 28]. Our proposal can adopt these mechanisms since fault detection is out of the scope of this work.

4 Proposed recovery method overview

Figure 2 exemplifies two possible situations handled by the proposed recovery method, using as example a 4×2 many-core instances, with 2×2 clusters. In Fig. 2a, $CM_{2,0}$ is faulty and $SP_{3,0}$ is free, i.e., there is no tasks executing on it. In this case, the proposed recovery method migrates the kernel (operating system) from $CM_{2,0}$ to $SP_{3,0}$. In Fig. 2b all SPs of the cluster managed by the faulty CM execute tasks. In this scenario, the recovery method migrates tasks executing in the cluster to another cluster, before migrating the kernel.

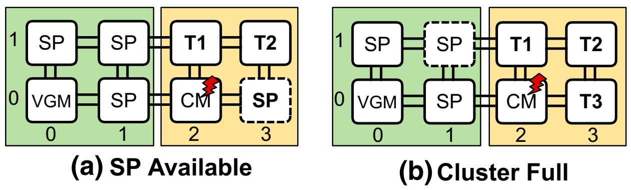

Fig. 2 Scenarios handled by the recovery method: **a** cluster with available SPs; **b** cluster with all SPs executing tasks.

The method defines at the system startup that two MPEs horizontally aligned are *MPEs pairs*. At runtime, an MPE may migrate to another position, changing the original configuration of the *MPE pair*. As there is no guarantee that the *MPE pair* continue to be physically aligned, the usage of broadcast messages enables the communication between the MPEs belonging to the pair. When a fault occurs, one MPE of the pair is healthy (Definition 2) and the other one is faulty (Definition 3).

Definition 2 MP_h—healthy MPE.

Definition 3 MP_f—faulty MPE.

Figure 3 overviews the recovery protocol. When a permanent fault is detected in an MPE, the processor is promptly isolated by wrappers to avoid Byzantine faults, and the control NoC notifies the fault to the MP_h by a broadcast control message. The MP_h starts the recovery method. It immediately injects a *freeze* message to all the

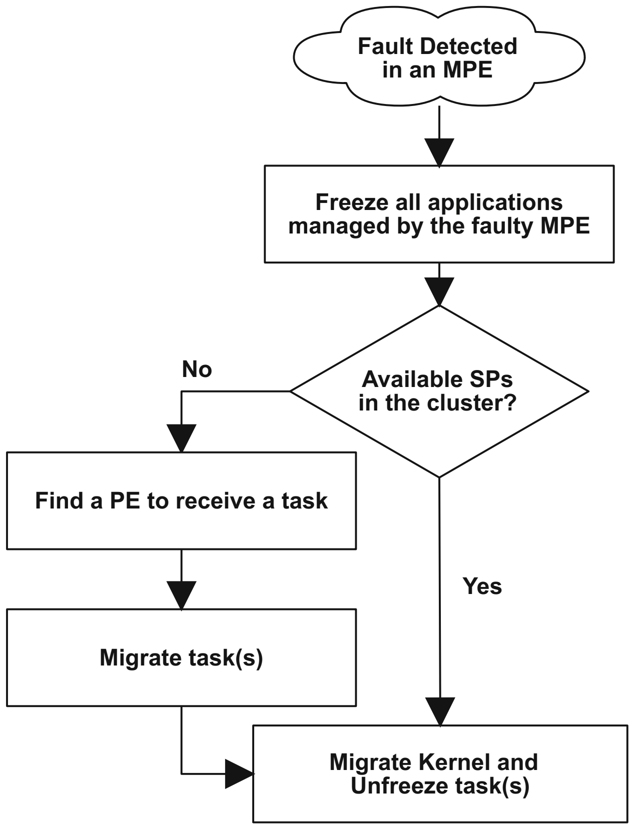
Fault Detected in an MPE
Freeze all applications managed by the faulty MPE
Available SPs in the cluster?
No
Yes
Find a PE to receive a task
Migrate task(s)
Migrate Kernel and Unfreeze task(s)

Fig. 3 High-level flow chart, with the actions executed by the recovery protocol

PEs. All tasks managed by MP_f stop their execution. Next, MP_h evaluates the PE location to receive the functions executed by MP_f. If there is an available SP in the cluster, i.e., with no tasks assigned to it, the kernel migration process starts. Otherwise, it is necessary to release an SP of the cluster managed by MP_f to another cluster. This action is done by migrating one or more tasks to a free SP. When the task migration finishes, the kernel migration begins. After the kernel migration, the PE that received the kernel assumes the role of the previous MP_f.

5 Actions executed by the recovery protocol

This section presents the actions executed by the recovery protocol. The criteria to select an SP_{cand} (Sect. 5.1), the method to freeze and unfreeze tasks (Sect. 5.2), the technique for task migration (Sect. 5.3) and the method to migrate the MPE memory contents (Sect. 5.4).

5.1 MPE candidate selection

At system startup, the closest SP to its MPE is the SP candidate—SP_{cand}, Definition 4. When the MPE maps a new application into the cluster, it verifies if the SP_{cand} has tasks assigned to it. In this case, the rule to select a new SP_{cand} is the SP with the minimum number of tasks assigned to it. Thus, after any application mapping, the MPE computes the SP_{cand} address and transmits it to its pair. The number of tasks executing in the SP_{cand} is also transmitted because if it's different from zero, the MPE pair will manage the migration of the tasks executing in the SP_{cand}.

Definition 4 SP_{cand}—an SP selected by its MPE to receive the management kernel, if this MPE fails.

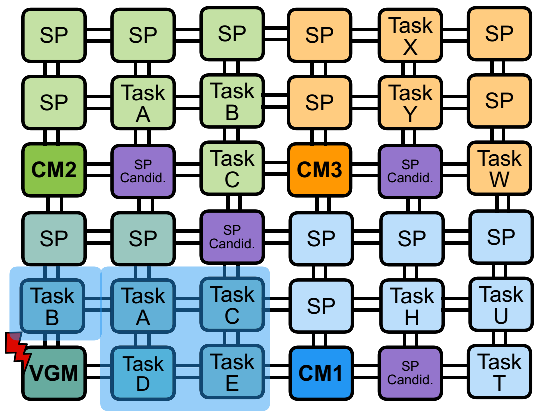

Fig. 4 Freeze process on the cluster managed by the *VGM*. CM1 injects the `freeze_message`

5.2 Freeze/unfreeze process

When an MPE fails, the tasks it manages are suspended to prevent control messages from being lost. This suspension process is called *freezing*. Since the MPE is faulty, its pair executes this action.

Freeze and unfreeze are control actions to stop or release the execution of a set of tasks. Figure 4 presents the freezing process that starts with a MP_h (CM1) transmitting in broadcast a `freeze_message`, by the control NoC, having in its payload the address of the MP_f (VGM). Any SP receiving a freeze message verifies if it has tasks managed by MP_f. In this case, all tasks of these SPs are freezed (blue region in the figure). Otherwise, the message is discarded. The transmission of the freeze message enables to stop tasks in SPs managed by MP_f executing in other clusters, due to the reclustering process. The freeze message does not stop the tasks immediately. To avoid messages losses, the task must be in a safe state. A safe state is defined as: the task to freeze should be ready to be scheduled by the kernel, and there is no pending request for messages. For example, if a task is in a waiting state, this means that the task requested a message to a producer task. Thus, the producer receives the request and at some moment inject messages into the NoC. Such procedure ensures that when a given task stops, there are no messages generated by the task in the data NoC. Thus, all tasks managed by MP_f goes to the freeze state, avoiding their scheduling by the kernel.

After the recovery process, the new MPE sends an `unfreeze_message`, also in broadcast. This message unfreezes the tasks managed by the new MPE and also transmits the new MPE address to the SPs of the cluster.

5.3 Task migration

The MP_h starts the task migration process. First, the MP_h sends a task migration message to the SP, notifying that the tasks it is running must migrate to a new PE—the *target PE*. Next, the SP operating system sends a set of messages to the target PE related to the tasks executing in the SP:

(i) Task code;
(ii) TCB (Task Control Block): a data structure that stores the task state, including the values stored at each register, PC (Program Counter), SP (Stack Pointer), size of the object code and data (for migration purposes);
(iii) Message Requests: a structure with the received requests for messages;
(iv) Stack data: data stored in the memory corresponding to the stack;

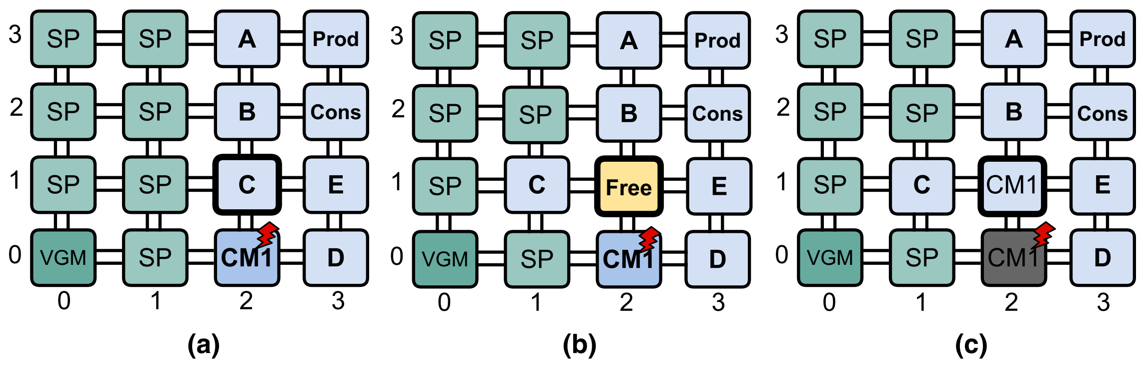

Fig. 5 Task migration to release an *SP*. **a** Fault detect at *CM*1; **b** task C migrated from $SP_{2,1}$ to $SP_{1,1}$; **c** *CM*1 migration from address (2, 0) to (2, 1)

(v) Pipe: all messages produced by the task but not yet delivered;

(vi) Data: includes the local data and BSS memory segments;

(vii) Tasks location: addresses of the tasks that communicate with the task being migrated.

The target PE after receiving all task messages related to the task migration execute the following actions: (1) sends a message to all SPs that communicate with the migrated task with the new task address; (2) sends a message to MP_h notifying that the migration process ended.

Figure 5 presents a possible scenario handled by the recovery protocol with task migration. In this scenario, it was detected a permanent fault at *CM*1. All SPs of this cluster have at least one task in execution, being $SP_{2,1}$ the SP_{cand}. In Fig. 5a occurs the fault detection at *CM*1. The wrapper control module (*WC*, in Fig. 1) notifies the fault by injecting a *fail_CPU_message* in the control NoC. The *VGM* knows that the SP_{cand} is executing task C. Thus, it is necessary a task migration before the recovery process. In Fig. 5b task C migrates from $SP_{2,1}$ to $SP_{1,1}$, in another cluster. When the task migration finishes, the kernel migrates to $SP_{2,1}$ (Fig. 5c).

5.4 Kernel migration

The kernel (operating system) migration differs from task migration. While in task migration it is possible to optimize the amount of data to be transmitted, the kernel migration requires the transmission of complete memory contents from the MP_f to the SP_{cand}.

Another difference between migration methods is the migration management. While in task migration the kernel itself executes this process, in the kernel migration this is not possible because the processor is faulty and isolated by wrappers. Thus, it was added in the *DMNI* module the ability to treat specific packets, which starts the process of transferring the memory contents.

The first step of the kernel migration process is to prepare the SP_{cand} to receive the MP_f memory contents (code and data). The MP_h, notifies the SP_{cand} that it will receive the kernel executing in MP_f, through a `wait_kernel_message`. A field in the packet header of this message defines that the *DMNI* module will process the message payload, not the processor. This message induces in the PE the following actions: (*i*) hold the processor and configure the *DMNI* module to write incoming packets into the memory, from address zero; (*ii*) after configuring the *DMNI* to write packets directly into the memory, the DMNI sends a `wait_kernel_acknowledge` message to MP_H.

Once received the `wait_kernel_acknowledge` message, the MP_h notifies MP_f to send the kernel to SP_{cand} through a `send_kernel_message`. The kernel migration is simpler than the task migration in the sense that only one message is transmitted with the complete memory contents.

An issue to discuss is how the recovery method affects the traffic in the NoC, which could penalize the performance of applications. A fault in an MPE (VGM or CM) affects applications running on the cluster managed by this MP_f. The MP_h stopped all applications managed by MP_f,

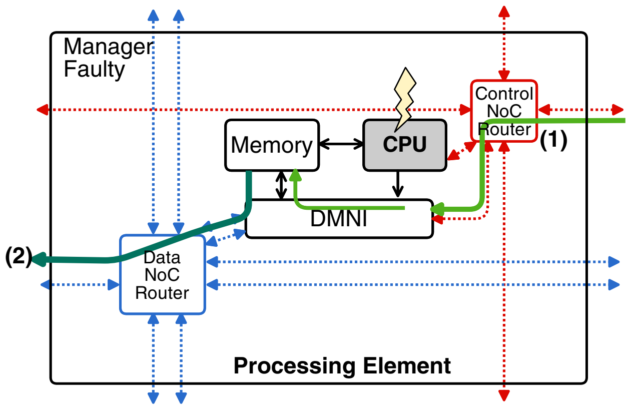

Fig. 6 Kernel migration process in an MP_f

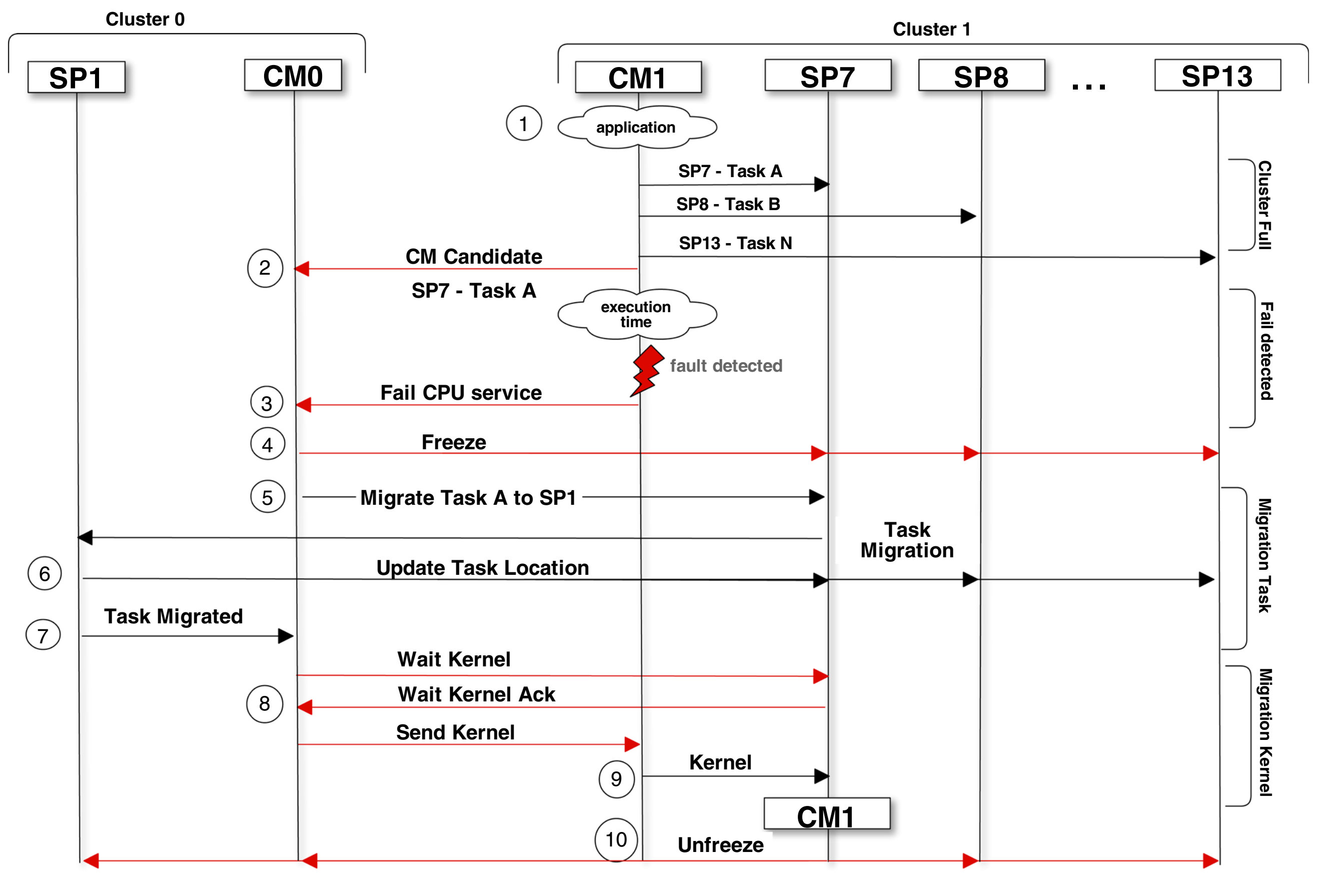

Fig. 7 Sequence diagram of the recovery protocol steps. Black arrows: messages transmitted through the Data NoC. Red arrow: messages transmitted through the Control NoC

but all other applications continue to run in their clusters, without being disturbed. A slight perturbation may occur during task migration. Consequently, the recovery method has a minimal impact in the NoC traffic, occurring only when it is necessary to migrate a task to a neighbor cluster.

Figure 6 presents how the MP_f handles the `send_kernel_message`. It is important to remind that the fault detection mechanism already isolated the CPU by wrappers. The *DMNI* of the MP_f handles this message (event **1** in the figure), transferring the memory contents to the SP_{cand} (**2**), using the data *NoC*.

6 Recovery protocol steps

Figure 7 presents the recovery method, assuming:

- A many-core with two clusters, being *CM*0 and *CM*1 the managers of clusters 0 and 1, respectively;
- SP7: SP_{cand}, executing 1 task;
- SP1: an idle processor from another cluster that will receive the task executing in SP7.

Cluster 1 receives application mapping requests (**1** in Fig. 7), assigning a task to each SP in its cluster. In this example, all SPs execute at least one task. After assigning the tasks in the cluster, SP7 is elected as a new SP_{cand}, and *CM*1 notifies *CM*0 that SP7 is the SP_{cand}, executing one task (**2**).

At a given moment (**3**), a permanent fault is detected in the processor of *CM*1. The control NoC receives the fault notification, and broadcast a *Fail CPU service* message, targeting the CM pair, in this case, *CM*0. The first action of the protocol, after the fault notification message, is to broadcast a freeze message (Sect. 5.2) to all tasks managed by *CM*1 (**4**).

The next protocol action is to migrate tasks, if necessary. In this example, it is necessary to migrate the task executing on SP7 to SP1. The *CM*0 sends a message to SP7 to migrate the task it is executing to SP1 (**5**). As detailed in Sect. 5.3, SP7 sends to SP1 a set of messages with the task contents. After receiving all messages related to the task migration, SP1 notifies to all application tasks the new location of the migrated task (**6**). The task migration ends with SP1 notifying the MP_h (*CM*0) the end of the migration process (**7**).

With the availability of a PE in the cluster, the kernel migration starts. Actions represented in event **8** correspond to the kernel migration protocol (Sect. 5.4): notification of the SP that will assume the CM role (SP7); the acknowledgment message to CM_H; and the message to transfer the memory contents from the CM_F to SP_{cand}. Next, the CM_F DMNI transfers the memory contents to SP_{cand} (**9**). Once the kernel received, the SP_{cand} restarts, assuming the role of a new CM. After restarting, the new CM sends an *unfreeze* message to the stopped task (**10**). This message unfreezes the tasks managed by the new *CM* and also transmits the CM address to the SPs.

Note that when the kernel is restarted, the processor knows that it is a restart from a migration. In this case, the contents of all data structures are preserved, without executing the kernel initialization.

7 Results

This section presents results related to the recovery protocol. Experiments are executed using a clock-cycle accurate RTL SystemC model of the reference many-core platform. Applications and kernel are described in C language, compiled from C code and executed over the platform model. The experiments adopt a 6×6 many-core instance, organized in 3×3 clusters. To evaluate the recovery protocol, five benchmarks execute in the MCSoC: MPEG decoder (5 tasks), Prod Cons (2 tasks), DTW (6 tasks), Synthetic (6 tasks) and Dijkstra (6 tasks).

This section evaluates the Workload Execution Time (*WET*) and the recovery method overheads, in milliseconds (@100 MHz). A common overhead in the experiments is the time required to migrate the kernel (64 KB), 1.5 ms (average value), and the time to migrate one task (10 KB, code and data), 0.3 ms (average value). These overheads vary proportionally with the kernel and task sizes. The reason to keep the same kernel and task sizes in the experiments comes from the fact that they do not impact in the remaining protocol steps.

Table 2 Overhead—VGM recovery

	Time (ms)
Fail CPU	2.8
Freeze	2.81
Wait Kernel	2.97
Unfreeze	4.32
WET (with recovery)	10.13
WET (baseline)	8.35

7.1 Recovery results from a fault in a manager

This section presents a scenario when the fault is injected in an MPE. This first evaluation corresponds to the best-case scenario for the protocol, since the SP_{cand} is free (without any task assigned to it).

Figure 8 presents the test-case to recover the VGM. Figure 8a presents the MCSoC state before the recovery method, being $SP_{2,2}$ the SP_{cand}. When the fault is detected by CM1 (VGM pair), the manager recovery method starts. Figure 8b presents the system state after the VGM migration to the SP_{cand}(2,2).

Table 2 details the time spent at each recovery protocol step. The 1st line contains the time when a fault was inserted and detected, 2.8 ms. The 2nd line shows the moment when the kernel migration starts. The 4th line corresponds to the moment when the recovery ended, 4.32 ms. The difference, 1.52 ms, is the delay mentioned above to migrate the kernel.

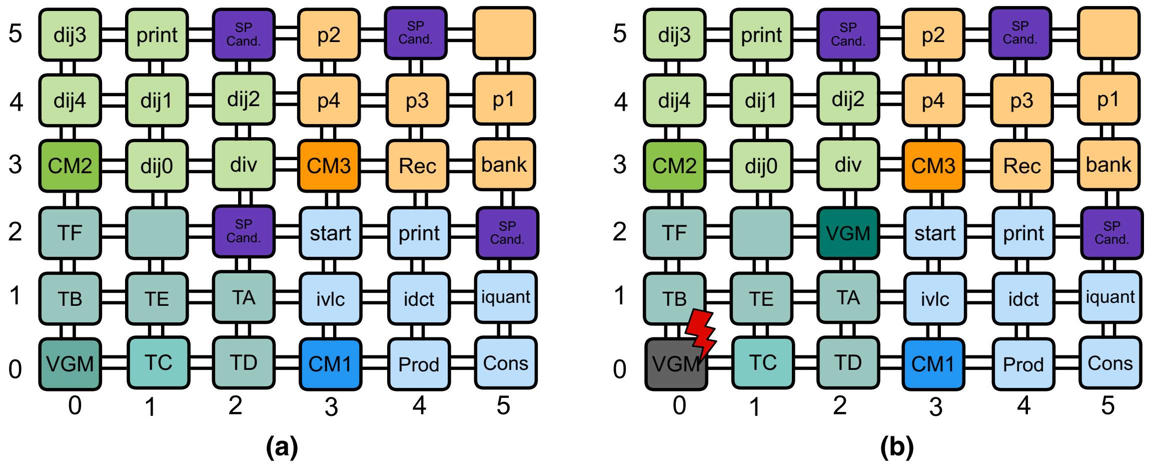

Fig. 8 Recovery method for the VGM

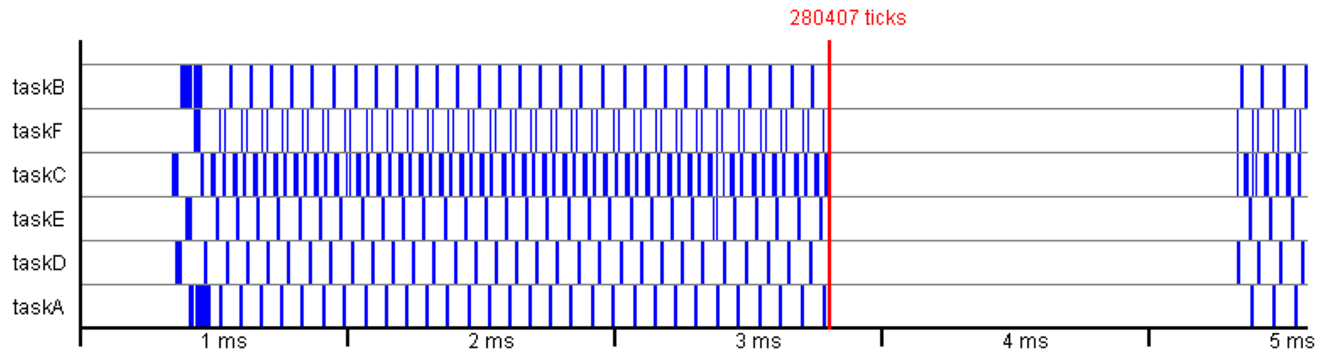

Fig. 9 Scheduling of the Synthetic tasks, showing the moment when the application is suspended

Table 3 Applications' execution time

Application	t_{start} (ms)	t_{end} (ms)
DTW	0.00	8.32
Synthetic	0.00	10.08
Dijkstra	1.50	7.98
Mpeg	2.54	7.04
Prod_cons	2.17	5.30

Table 5 Applications' execution time

Application	t_{start} (ms)	t_{end} (ms) B–R
MPEG-1	2.00	7.02–8.54
Synthetic	0.00	9.05–9.05
Dijkstra	1.50	7.09–7.09
MPEG-2	0.00	5.06–5.06

The last two lines correspond to the *WET*, with (5th) and without (6th) the recovery method. The difference (1.78 ms) is slightly higher than the kernel recovery time. The reason for explaining the increase in the total execution time is mainly due to the rescheduling of tasks. Figure 9 illustrates the task scheduling of the Synthetic application, which was running in the left-most cluster. It is possible to observe the moment when all tasks were suspended due to the freeze message, and later the moment of reactivation (unfreeze message). Given the interdependence between the tasks, there is an overhead for the resynchronization between them.

It is worthwhile to mention that there is no relationship between the number of tasks and the WET. The additional overhead (0.26 ms in the experiment) is due to the resynchronization between the tasks of the applications affected by the freezing process during the kernel recovery.

Table 4 Overhead—CM recovery

	Time (ms)
Fail CPU	2.5
Freeze	2.51
Wait Kernel	2.67
Unfreeze	4.02
WET (R—with recovery)	9.06
WET (B—baseline)	9.06

Table 3 shows the time when each application starts and ends its execution. Note that all applications were executing when the fault was injected into the VGM (2.8 ms). The t_{start} in Table 3 corresponds to the moment that application should be deployed into the MCSoC. The VGM executes the cluster selection, in the sequence occurs the task mapping, the transmission of the object code of the tasks to the SPs, and finally, the task is scheduled. Thus, even if $t_{start} = 0$, as for the Synthetic application, this application actually starts at 0.5 ms.

Results to recover a CM is similar to the VGM recovery, 1.52 ms to migrate the CM1 kernel. Table 4 details the time spent at each recovery protocol step. Table 5 shows the time when each application starts and ends its execution. The *WET*, with and without recovery, is the same. The reason for explaining the same *WET* is that the application affected during the recovery method (MPEG-1) finishes its execution before the Synthetic application (9.05 ms). The overhead occurs only in the MPEG-1 execution time, 5.02 ms to 6.64 ms, resulting in an overhead equal to 1,62 ms. This result presents an advantage of the method, which is the fact the recovery method overhead can be masked if the set of applications executing on the MP_h clusters takes longer to execute than the applications executing on the MP_f cluster.

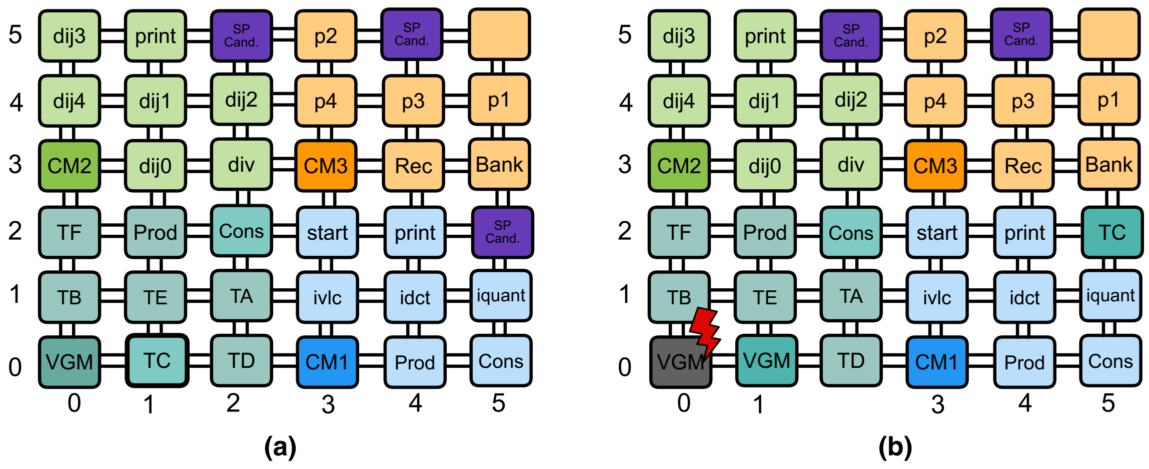

Fig. 10 Recovery method for the VGM and a task migration

In both scenarios, with faults injected into the VGM or CM, the overhead is the time to migrate the memory contents (code and data of the MP_f) to the SP_{cand}. When an MPE fails, the tasks it manages should be suspended to prevent control messages from being lost (freeze), delaying applications. For both VGM or CM recovery, the overhead was the same, corresponding to 1.5 ms@100 MHz, or 150,000 clock cycles.

7.2 Recovery results from a fault in an mpe with task migration

This section presents a scenario when the fault is injected in an MPE, and the cluster has all resources in use. Thus, the SP_{cand} is not free. i.e., it has tasks assigned to it, being necessary to execute task migration before the recovery method starts.

Figure 10 shows the test case to recover the VGM, executing task migration before kernel migration. Figure 10a presents the MCSoC state before the recovery protocol, being $SP_{1,0}$ the SP_{cand}. When the fault is detected by CM1 (VGM pair), the manager recovery method starts. The task *TC* migrates from $SP_{1,0}$ to $SP_{5,2}$. After task migration, the $SP_{cand}(1{,}0)$ receives the VGM kernel. Figure 10b presents the system state after the VGM kernel migration.

Figure 11 illustrates the task scheduling of *TC*, which was allocated and running in $SP_{1,0}$ up to 3.0 ms, and migrated to $SP_{5,2}$. It is possible to observe when the task was suspended due to the freeze message in $SP_{1,0}$ at 3.0 ms, migrated, and later the moment of reactivation (unfreeze message) in the $SP_{5,2}$ at the 4.5 ms.

Table 6 Overhead—VGM recovery and task migration

	Time (ms)
Fail CPU	3.00
Freeze	3.01
Migration	3.29
Wait Kernel	3.30
Unfreeze	4.65
WET (with recovery)	9.92
WET (baseline)	8.61

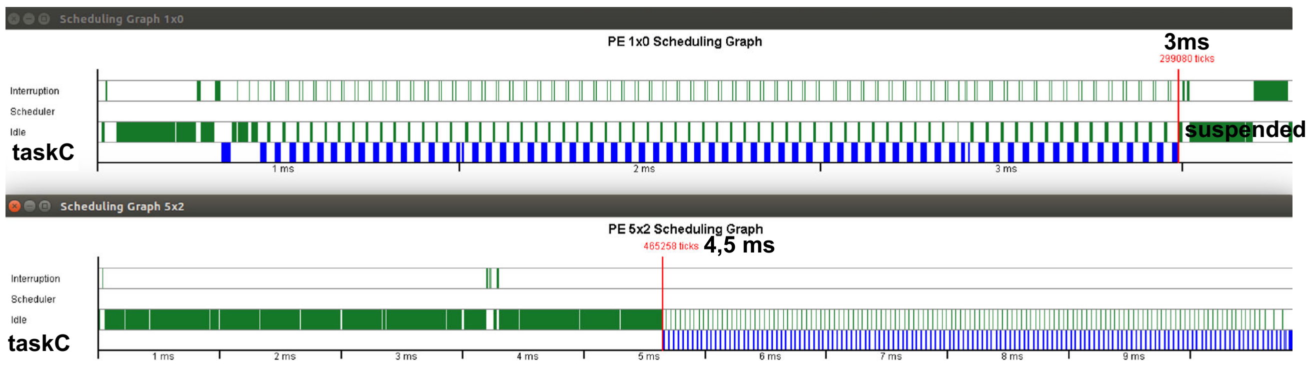

Fig. 11 Scheduling of Task C, showing the moment when the task migrate

Table 7 Applications' execution time

Application	t_{start} (ms)	t_{end} (ms)
DTW	0.00	8.33
Synthetic	0.30	9.89
Dijkstra	1.50	7.11
Mpeg	2.50	7.04
Prod_cons	2.17	5.24
Prod_cons	2.00	7.01

Table 8 Overhead—CM recovery and task migration

	Time (ms)
Fail CPU	3.00
Freeze	3.01
Migration	3.29
Wait Kernel	3.30
Unfreeze	4.65
WET (with recovery)	11.86
WET (baseline)	10.20

Table 9 Applications' execution time

Application	t_{start} (ms)	t_{end} (ms)
Synthetic	2.50	11.82
Mpeg	0.03	5.07
DTW	0.00	8.21
Dijkstra	1.50	7.10
Prod_cons	2.17	5.29
Prod_cons	2.00	5.12
Prod_cons	2.00	7.02

Table 6 details the time spent at each recovery protocol step. The 1st and 2nd lines present when the fault was inserted and detected, 3.0 and 3.01 ms, respectively. The 3rd line shows the moment when task migration ended. The 4th line shows the moment when kernel migration starts. The 5th line shows the moment when the recovery ended. Table 7 shows the time when each application starts and ends its execution.

The overhead induced by the recovery method and a task migration was 1.65 ms. However the *WET* with recovery presents an overhead equal to 1.31 ms. The overhead is lower than expected due to the fact that task *TC* is originally in a position with high data traffic, and with migration, its mapping reduced the network congestion. This experiment shows that a reduced number of hops between tasks, the primary function of the mapping heuristics, may impact negatively in the application performance.

Table 8 details the time spent at each recovery protocol step when CM1 fails. The 1st and 2nd lines present when the fault was inserted and detected, 3.0 and 3.01 ms, respectively. The 3rd line shows the moment when task migration ended. The 4th line shows the moment when kernel migration starts. The 5th line shows the moment when the recovery ended. Table 9 shows the time when each application starts and ends its execution. In this experiment, the overhead induced by the recovery method and a task migration was 1.65 ms, and the *WET* overhead 1.66 ms. They are, in practice, the same because the affected application by the CM1 fault is the one with the longest execution time.

In both VGM or CM1 fault scenarios, the overhead is the time spend to migrate the memory contents (code and data of the MP_f) to the SP_{cand} and the task migration. When an MPE fails, the tasks it manages should be suspended to prevent control messages from being lost (freeze). The freezing process delays the application. For both VGM or CM1 recovery and the task migration, the time overhead was 1.65 ms or 165,000 clock cycles.

The requirement to initiate the recovery method is to select a free SP to become a new manager. The SP candidate can have tasks assigned to it, and the cost to free the SP is the migration cost.

7.3 Final remarks

Table 10 summarizes the results presented in this section. It is possible to state that a fault in an MPE induces a runtime overhead of around 1.5 ms (150,000 clock cycles), and it increases according to the size of the kernel memory footprint.

Table 10 Summary of results

Fault Location	Relevant protocol feature	Protocol overhead (kernel: 64KB/task: 10 KB) (ms)
VGM CM	Without task migration	1.5
VGM CM	With task migration	1.65

The evaluation made in this section focused on the method overhead in terms of performance. There are two implementation costs: software and hardware. The cost of the software refers to the increase in memory required by the kernels running on *VGM*/*CM*, from 12 to 43 KB, and on SP kernel, from 19 to 34 KB.

The hardware costs associated with the methods can be listed as follows: (*i*) control NoC network, area equivalent to 20% of a data network router; (*ii*) wrappers, it require only logic gates to isolate control signals; (*iii*) it is assumed that the memory is protected by *ECC* (error-correcting codes) and that the network interface has access to this memory in case of processor fault. Therefore, the hardware cost is minimal, being portable for other MCSoC architectures.

8 Conclusion

This work presented a runtime protocol for management recovery in NoC-based many-core. The proposal includes a method to safely migrate the management software to a new processing element, assuming a protected memory and a task migration method to release an SP candidate. The results displayed a small overhead for the task migration, as well as a small impact on the execution time of the applications when they are stopped to migrated the management functions to another PE (1.5 to 1.65 ms).

Future works include: (*i*) extend the method to faults in slave processing elements, enabling to recover applications from faults; (*ii*) add multiple interfaces to the external environment to avoid a single point of failure, i.e., enable multiple *CMs* to receive application requests.

References

1. Barreto, F., Amory, A. M., & Moraes, F. G. (2015). Fault recovery protocol for distributed memory MPSoCs. In *IEEE international symposium on circuits and systems (ISCAS)* (pp. 421–424).
2. Bhowmik, B., Deka, J. K., Biswas, S., & Bhattacharya, B. (2016). On-line detection and diagnosis of stuck-at faults in channels of NoC-based systems. In *IEEE international conference on systems, man, and cybernetics (SMC)* (pp. 4567–4572).
3. Bolchini, C., Carminati, M., & Miele, A. (2013). Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing: Theory and Applications*, *29*(2), 159–175.
4. Boraten, T., & Kodi, A. K. (2018). Runtime techniques to mitigate soft errors in network-on-chip (NoC) architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(3), 682–695.
5. Braak, T. D. T., Burgess, S. T., Hurskainen, H., Kerkhoff, H. G., Vermeulen, B., & Zhang, X. (2010). On-line dependability enhancement of multiprocessor SoCs by resource management. In *SoC* (pp. 103–110).
6. Brillu, R., Pillement, S., Lemonnier, F., & Millet, P. (2013). Cluster based MPSoC architecture: an on-chip message passing implementation. *Design Automation for Embedded Systems*, *17*(3–4), 587–607.
7. Caimi, L., Fochi, V., Wachter, E., Munhoz, D., & Moraes, F. G. (2017). Secure admission and execution of applications in many-core systems. In *Symposium on integrated circuits and systems design (SBCCI)* (pp. 65–71).
8. Carara, E., de Oliveira, R., Calazans, N., & Moraes, F. G. (2009). HeMPS—A framework for NoC-based MPSoC generation. In *IEEE international symposium on circuits and systems (ISCAS)* (pp. 1345–1348).
9. Castilhos, G., Mandelli, M., Madalozzo, G., & Moraes, F. G. (2013). Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *IEEE computer society annual symposium on VLSI (ISVLSI)* (pp. 153–158).
10. Chen, Y., Chang, E., Hsin, H., Chen, K., & Wu, A. (2017). Path-diversity-aware fault-tolerant routing algorithm for network-on-chip systems. *IEEE Transactions on Parallel and Distributed Systems*, *28*(3), 838–849.
11. Domingues, A. R. P., Hamerski, J. C., & Amory, A. (2018). Broker fault recovery for a multiprocessor system-an-chip middleware. In *Symposium on integrated circuits and systems design (SBCCI)* (pp. 1–6).
12. Dutt, N., Jantsch, A., & Sarma, S. (2015). Self-aware cyber-physical systems-on-chip. In *IEEE/ACM international conference on computer-aided design (ICCAD)* (pp. 46–50).
13. Fick, D., DeOrio, A., Hu, J., Bertacco, V., Blaauw, D., & Sylvester, D. (2009). Vicis: A reliable network for unreliable silicon. In *DAC* (pp. 812–817).
14. Fochi, V., Wächter, E., Erichsen, A., Amory, A. M., & Moraes, F. G. (2015). An integrated method for implementing online fault detection in NoC-based MPSoCs. In *IEEE International symposium on circuits and systems (ISCAS)* (pp. 1562–1565).
15. Heron, O., Guilhemsang, J., Ventroux, N., & Giulieri, A. (2010). Analysis of on-line self-testing policies for real-time embedded multiprocessors in DSM technologies. In *IEEE international conference on electronics, circuits and systems (ICECS)* (pp. 49–55).
16. Kamran, A., & Navabi, Z. (2016). Stochastic testing of processing cores in a many-core architecture. *Integration, the VLSI Journal*, *55*(1), 183–193.
17. Kim, H., Vitkovskiy, A., Gratz, P. V., & Soteriou, V. (2013). Use it or lose it: Wear-out and lifetime in future chip multiprocessors. In *IEEE/ACM international symposium on microarchitecture (MICRO)* (pp. 136–147).
18. Martins, A. L. M., Sant'Ana, A. C., & Moraes, F. G. (2016). Runtime energy management for many-core systems. In *IEEE international conference on electronics, circuits and systems (ICECS)* (pp. 380–383).
19. Meloni, P., Tuveri, G., Raffo, L., Cannella, E., Stefanov, T. P., Derin, O., Fiorin, L., & Sami, M. (2012). System adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project approach. In *Euromicro conference on digital system design (DSD)* (pp. 517–524).
20. Paul, J., Oechslein, B., Erhardt, C., Schedel, J., Kröhnert, M., Lohmann, D., et al. (2015). Self-adaptive corner detection on MPSoC through resource-aware programming. *Journal of System Architecture*, *61*(10), 520–530.
21. Reddy, B., Vasantha, M., & Kumar, Y. (2016). A gracefully degrading and energy-efficient fault tolerant NoC using spare core. In *IEEE computer society annual symposium on VLSI (ISVLSI)* (pp. 146–151).
22. Ruaro, M., Lazzarotto, F. B., Marcon, C. A., & Moraes, F. G. (2016). DMNI: A specialized network interface for NoC-based

MPSoCs. In *IEEE international symposium on circuits and systems (ISCAS)* (pp. 1202–1205).
23. Silveira, J., Marcon, C., Cortez, P., Barroso, G., Ferreira, J. M., & Mota, R. (2016). Scenario preprocessing approach for the reconfiguration of fault-tolerant NoC-based MPSoCs. *Microprocessors and Microsystems*, *40*(1), 137–153.
24. Paul, S., Chatterjee, N., & Ghosal, P. (2018). A permanent fault tolerant dynamic task allocation approach for network-on-chip based multicore systems. *Journal of Systems Architecture*, *97*(1), 287–303.
25. Tajik, H., Donyanavard, B., Dutt, N., Jahn, J., & Henkel, J. (2016). SPMPool: Runtime SPM management for memory-intensive applications in embedded many-cores. *ACM Transactions on Embedded Computing Systems*, *16*(1), 25:1–25:27.
26. Tsoutsouras, V., Masouros, D., Xydis, S., & Soudris, D. (2017). SoftRM: Self-organized fault-tolerant resource management for failure detection and recovery in NoC based many-cores. *ACM Transactions on Embedded Computing Systems*, *16*(5s), 144:1–144:19.
27. Wachter, E., Caimi, L. L., Fochi, V., Munhoz, D., & Moraes, F. G. (2017). BrNoC: A broadcast NoC for control messages in many-core systems. *Microelectronics Journal*, *68*(1), 69–77.
28. Walters, J. P., Kost, R., Singh, K., Suh, J., & Crago, S. P. (2011). Software-based fault tolerance for the Maestro many-core processor. In *IEEE aerospace conference* (pp. 1–12).
29. Wentzlaff, D., et al. (2007). On-chip interconnection architecture of the tile processor. *IEEE Micro*, *27*(5), 15–31.
30. Yu, Q., Zhang, M., & Ampadu, P. (2011). Exploiting inherent information redundancy to manage transient errors in NoC routing arbitration. In *NoCS* (pp. 105–112).
31. Zhang, Y., Morris, R., DiTomaso, D., & Kodi, A. (2012). Energy-efficient and fault-tolerant unified buffer and bufferless crossbar architecture for NoCs. In *IPDPS* (pp. 972–981).

Vinicius Fochi received the Computer Engineer in 2012 and the M.Sc. in 2015 degrees from the Pontifical Catholic University of Rio Grande do Sul (PUCRS). He is now Ph.D. student in the same University. His main research interest includes multiprocessor systems on chip (MPSoCs), and fault-tolerant systems.

Luciano L. Caimi received the M.Sc. degree in Electrical Engineer from Federal University of Santa Catarina (UFSC) in 1998. He is currently Assistant Professor at Federal University of Fronteira Sul (UFFS) and Ph.D. student at PUCRS University. His main research interest includes multiprocessor systems on chip (MPSoCs), and security for embedded systems.

Marcelo H. da Silva is an undergraduate student at the Computer Engineering course at PUCRS, Porto Alegre, Brazil. His main research interest includes multiprocessor systems on chip (MPSoCs), and NoCs.

Fernando Gehm Moraes received the Electrical Engineering and M.Sc. degrees from the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1987 and 1990, respectively. In 1994 he received the Ph.D. degree from the Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier), France. He is currently at PUCRS, where he has been an Associate Professor from 1996 to 2002, and Full Professor since 2002. He has authored and co-authored 32 peer refereed journal articles in the field of VLSI design. His primary research interests include Microelectronics, FPGAs, reconfigurable architectures, NoCs and MPSoCs.