Fault-tolerance at the Management Level in Many-core Systems

Vinicius Fochi*, Luciano L. Caimi*†, Marcelo H. da Silva*, Fernando Gehm Moraes*
*PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
†UFFS – Av. Fernando Machado 108E, 89802-112, Chapecó, Brazil
vinicius.fochi@acad.pucrs.br, lcaimi@uffs.edu.br, marcelo.holgado@acad.pucrs.br, fernando.moraes@pucrs.br

***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 processing element reserved exclusively to execute such actions. A centralized approach would induce a significant load to the managers PE (MPE) in large-scale systems. The adoption of distributed approaches, with MPEs hierarchically organized, reduces the management load, being the organization adopted in this work. Consider a permanent fault in an MPE. In a centralized approach the entire system is compromised, and in a hierarchical organization, a set of PEs become inaccessible due to the fault in the MPE responsible for managing these PEs. 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 methods 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 protocol 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 executing in the many-core, with a small execution overhead observed during the management and task migration.**

***Index Terms*—Many-core; NoC; System Management; Fault-recovery protocol; Task migration; Fault-tolerance.**

I. Introduction

Modern many-core systems enable embedded systems with dozens of processors. Networks-on-Chip (NoCs) provide enhanced performance and scalability for the communication infrastructure. However, large systems demand one or more dedicated Processing Element (PE) for management actions, as task mapping/migration, power management (DVFS), QoS management, self-awareness adaptation, system monitoring (energy, temperature, deadlines). How to deal with a failure on Manager PEs (MPEs) opens a set of new challenges and opportunities in the field of many-core systems research.

The management of a many-core may be centralized or hierarchically organized in clusters (Definition 1) [1], [2]. In a centralized approach, if the MPE presents a permanent fault, the entire system halts. In the hierarchical organization when a MPE became unresponsive, only a many-core region is affected.

Definition 1. *Cluster* - a many-core virtual region, with a set of PEs and one MPE, named *Cluster Manager* (CM). 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 [1].

978-1-5386-7431-4/18/$31.00

With the ever-increasing reduction in the devices geometry, transistors, vias, and wires degrade faster over time, causing transient and permanent failures to occur earlier, thereby decreasing the lifetime of integrated circuits [3]. For these reasons, reliability becomes a critical design issue in many-core systems [4]. Classical fault-tolerant approaches, as Triple Module Redundancy (TMR) or spare components [5], do not comply with today's requirements of silicon area and power dissipation. By construction, a many-core contains a set of replicated structures (PEs), where a healthy component can replace the faulty component functions, with minimal performance reduction.

Thus, many-cores need management mechanisms to perform different control tasks at the *system leve*l, and the technology evolution from one side enables to increase the number of PEs and from the other side accelerates the emergent of failures. The literature presents different approaches at the *system level*: power management (e.g., DVFS), performance and QoS management, resource management and security [6], [7], [8], [9]. A rich literature with methods to test PEs is also available, with approaches adopted at different levels (hardware or software) or modules (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 system management requires monitoring and actuation policies to recover the system when one of these processors presents a permanent fault.

This work adopts many-core architectures with a hierarchical organization, with clusters. The paper *goal* is to present a recovery method when a CM became faulty, and propose 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 CM 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 II presents related work. Section III details the system architecture and the fault model. Section IV overviews the recovery method. Section V presents the actions executed by the recovery protocol. Section VI details the recovery protocol steps. Section VII presents the results, and Section VIII concludes this paper.

II. Related Work

Paul et al. [10] propose a resource-aware computing paradigm called Invasive Computing to reduce the negative effects of resource sharing in MPSoCs with the focus in mobile robotics applications. In the proposal, the operating system can influence the internal decisions in the application based on dynamic load distribution. Using the resource-aware programming model the application gain the ability to adapt to available resources by changing their workload. The model helped to avoid frame drops. No frame was dropped during the evaluation, and the accuracy values improved significantly over the conventional approach.

Chen et al. [11] propose a Path-Diversity-Aware Fault-Tolerant Routing (PDA-FTR) algorithm for NoCs. The PDA-FTR combines adaptive path and routing path quality to achieve fault-resilient packet delivery (FPD) and traffic load distribution. In the proposal, the algorithm uses the PDA information and a local buffer occupancy to acquire the Effective Buffer Length (EBL) of the routing direction. EBL is a router delay measurement, where higher EBL implies shorter routing delay. The routing decision made by the proposed routing algorithm sends the packet to a less congested region and away from the faulty region. To reduce the cost of memory, the Authors propose a regional FPD table to cover most routing paths for packet transmission while minimizing performance degradation. Due to the data locality, processing elements that often communicate with each other are usually placed in proximity to each other for minimizing the delay in data delivery.

Kamran et al. [12] propose an autonomous test mechanism for online detection of permanent faults in many-core processors. In this method, several test components are incorporated in the many-core architecture that autonomously and concurrent with the system normal operation, distribute software-based self-test routines among the processing cores, monitor the behavior of the processing cores during execution of the test routines, detect faulty cores, and make their omission from the system possible. To use short idle times of the processing cores, test data is segmented into small pieces, called test snippets. Individual test snippets are distributed among the processing cores and are made accessible to them for a limited period. If a processing core has an idle slot during a period that a test snippet is available, it executes the test snippet. Otherwise, it skips execution of that portion of the test.

Bhowmik et al. [13] present a distributed online test mechanism that detects stuck-at faults (SAFs) in the channels as well as identifies the faulty channel-wires in an on-chip network (NoC). The proposed test mechanism improves yield and reliability of NoCs at the cost of few test clocks and small performance degradation. The method focus on detecting stuck-at-0 (SA0) and stuck-at-1 (SA1) faults in the channels and evaluate their effect on network performance. Each channel consists of control, data, and handshake wires. The Authors propose an on-line method for testing channel-wires connecting a node (router and its core).

Tsoutsouras et al. [14] 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 failure 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: (*i*) controller cores: responsible for monitoring the system status sending this data to the set of the PEs; (*ii*) manager core: responsible for managing one application; (*iii*) worker cores: execute the applications' tasks. The cluster area is monitored by a controller core defined at system startup, but cannot change at runtime.

Previous works 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 CM to a healthy PE.

III. System Architecture and Fault Model

Figure 1 presents the reference many-core platform. It is an adaptation of the public-available HeMPS many-core [15]. The architecture contains a set of PEs interconnected by a *data NoC* and a *control NoC* [16]. One CM has an interface with the external environment to the many-core to receive new applications. Each PE contains one processor (32 bits MIPS), a Direct Memory Network Interface (DMNI, combining the functions of a Network Interface and a DMA module) [17], a dual-port private memory, the data and control routers. The PEs hardware is the same, being the role assigned to the PEs made by software: Slave PEs (SPs) executes users' tasks, supporting multitasking and message exchanging; CM manage a given cluster.

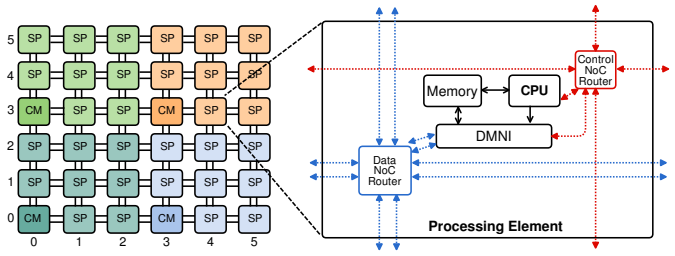

Fig. 1. A 6x6 reference many-core system instance with 3x3 clusters.

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

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

The *control NoC* [16] transfers the control messages. The current work uses the *control NoC* to transmit messages with the following purposes:

- notify the status of the CMs;
- freeze application(s) managed by a given CM;
- notify an SP that it will become a new CM;
- notify a DMNI module to transfer the memory contents of an SP to a new system address;
- notity the SPs about a new CM address;
- unfreeze application(s) after the CM migration.

An important architecture 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, transfer the memory contents when the processor has a permanent fault, is commonly adopted in fault-tolerant approaches [18].

The method herein presented may be applied to homogeneous or heterogeneous many-cores. The proposed method requires the following architectural features: (*i*) a set of PEs with the same architecture; (*ii*) at least two disjoint NoCs, one for application data and one for management purposes [19]; (*iii*) a memory module that can be read/write directly by the network interface [18].

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. 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 [20], and according to the transient faults severity, trigger the proposed protocol.
- *PE faulty module*: CPU. The proposed method is fired a when a permanent fault is detected in a CM. A fault detection mechanisms (out of scope) detects faults in the CPU and inject a message to report it.

IV. Proposed Recovery Method Overview

Figure 2 exemplifies two possible situations handled by the proposed recovery method, using as example a 4x2 many-core instances, with 2x2 clusters. In Figure 2(a), $CM_{2,0}$ 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 from $CM_{2,0}$ to $SP_{3,0}$. In Figure 2(b) 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.

The proposal starts by defining *CM pairs*. Each CM selects its pair at runtime. A pair of CMs is responsible for supervising each other, by exchanging periodically control messages, or for receiving a fault message.

Figure 3 presents the actions taken when a CM presents a permanent fault. When a permanent fault is detected in a CM (faulty CM, or CM_F), its CM pair (healthy CM, or CM_H) is notified by a broadcast control message. The CM_H starts the recovery method. The CM_H immediately inject a *freeze* message to all the PEs. All tasks managed by CM_F stop their

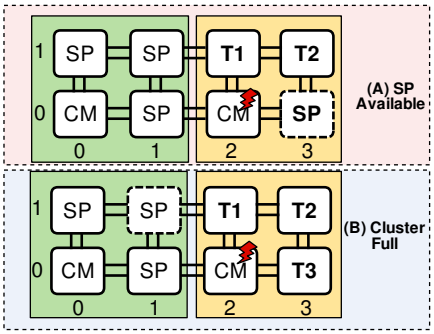

Fig. 2. Scenarios handled by the recovery method: (a) cluster with available SPs; (b) cluster with all SPs executing tasks.

execution. Next, CM_H evaluate the PE location to receive the functions executed by CM_F. If there is an available SP in the cluster, i.e., with no tasks assigned it, the kernel migration process starts. Otherwise, it is necessary to release an SP of the cluster managed by CM_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 CM_F.

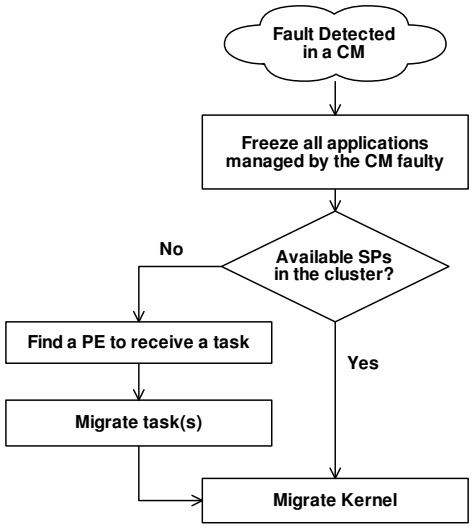

Fig. 3. High-level flow chart, with the actions executed by the recovery protocol.

V. Actions Executed by the Recovery Protocol

This Section presents the actions executed by the recovery protocol. The criteria to select the new $CM_{candidates}$ (Section V-A), the method to freeze and unfreeze tasks (Section V-B), the technique for task migration (Section V-C) and the method to migrate the CM memory contents (Section V-D).

A. CM Candidate Selection

At system startup, the closest SP to its CM is the $CM_{candidate}$. When the CM maps a new application into the cluster, it verifies if $CM_{candidate}$ has tasks assigned to it. In

this case, the rule to select a new $CM_{candidate}$ is the SP with the minimum number of tasks assigned to it. Thus, after any application mapping, the CM computes the $CM_{candidate}$ address and transmits it to its CM pair. The number of tasks executing in $CM_{candidate}$ is also transmitted because if its different from zero, the CM pair will manage the migration of the tasks executing in $CM_{candidate}$ if CM fails.

B. Freeze/Unfreeze Process

Freeze and unfreeze are control actions to stop or release the execution of a set of tasks. The freezing process starts with a healthy CM (CM_H) transmitting in broadcast a *freeze* message, using the Control NoC, having in its payload the address of the faulty CM (CM_F). It is necessary to stop (freeze) the tasks to prevent loss of control messages.

Any SP receiving a *freeze* message verifies if it has tasks managed by CM_F. If it is the case, all tasks of this SP must freeze. Otherwise, the message is discarded. The broadcast transmission of the *freeze* message enables to stop tasks in SPs managed by CM_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 CM_F goes to the *freeze state*, avoiding their scheduling by the kernel.

After the kernel migration, the new CM sends an *unfreeze* message, also in broadcast. This message unfreezes the tasks managed by the new CM and also transmits the new CM address to the SPs of the cluster.

C. Task Migration

The CM_H starts the task migration process. The CM_H sends to the SP with the task(s) to migrate a set of messages to migrate the structures related to the task:

(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;
(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: (*i*) sends a message to all SPs that communicate with the migrated task with the new task address; (*ii*) sends a message to CM_H notifying that the migration process ended.

D. Kernel Migration

The kernel migration process first step is to prepare the $CM_{candidate}$ to receive the memory contents (code and data) of CM_F. The CM_H notifies the $CM_{candidate}$ that it will receive the kernel executing in CM_F, through a *wait kernel* message. A special field in the packet header of this message defines that the DMNI module should 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*) once the DMNI configured to write packets directly into the memory, the DMNI sends a *wait kernel acknowledge* message to CM_H.

Once received the acknowledgement, CM_H sends a message to CM_F: *send kernel*. The kernel migration is simpler than the task migration because the kernel has no context (it is not controlled by a scheduler), neither communication structures (as requests and pipe). The DMNI of the CM_F handles the *send kernel* message, transferring the memory contents (code and data) to $CM_{candidate}$, using the data NoC. After transferring the memory contents, the DMNI is configured to avoid any transmission from the faulty processor, preventing Bizantine faults.

VI. Recovery Protocol Steps

Figure 4 presents the recovery method, assuming:

– a many-core with two clusters, being $CM0$ and $CM1$ the managers of clusters 0 and 1, respectively;
– SP7: $CM_{candidate}$, 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 Figure 4), 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 $CM_{candidate}$, and $CM1$ notifies $CM0$ that SP7 is the $CM_{candidate}$, executing one task (**2**).

At a given moment (**3**), a permanent fault is detected in the processor of $CM1$. The control NoC receives the fault notification, and broadcast a *Fail CPU service* message, targeting the CM pair, in this case, $CM0$. The first action of the protocol, after the fault notification message, is to broadcast a freeze message (Section V-B) to all tasks managed by $CM1$ (**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 $CM0$ sends a message to SP7 to migrate the task it is executing to SP1 (**5**). As detailed in Section V-C, SP7 sends to SP1 a set of messages with the task contents. After receiving all the 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 healthy CM ($CM0$) the end of the migration process (**7**).

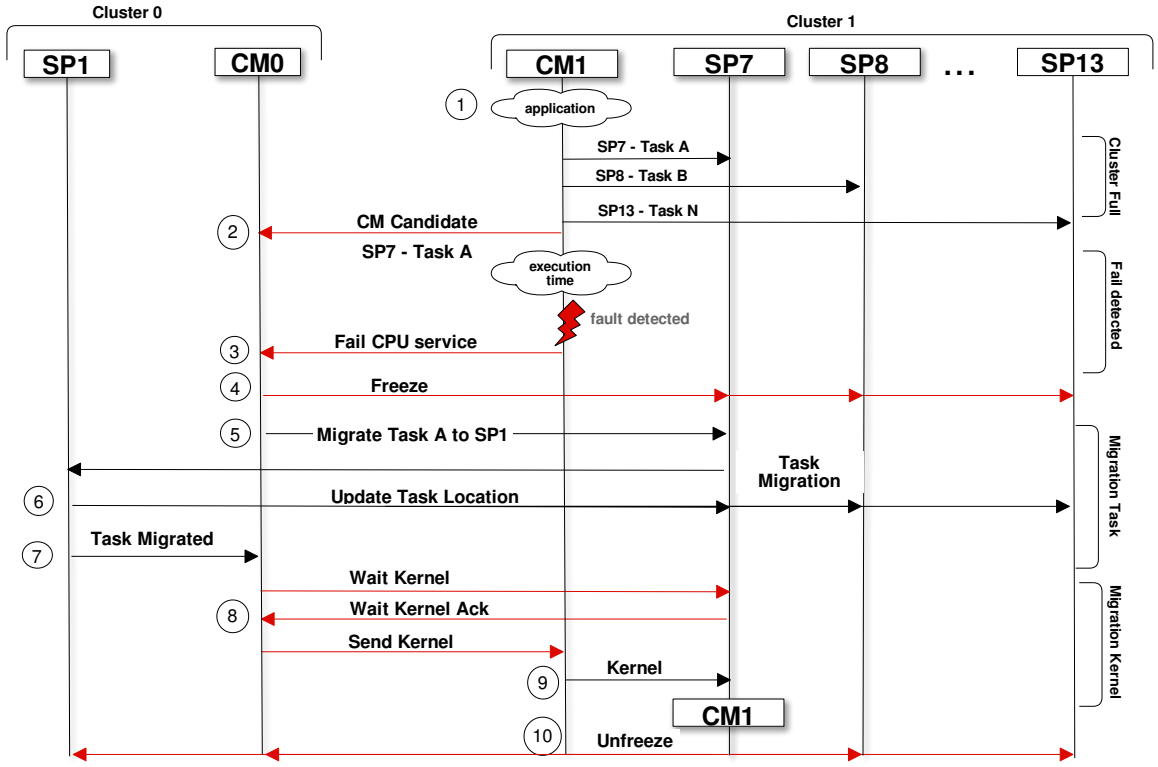

Fig. 4. Sequence diagram of the recovery protocol steps. Black arrows: messages transmitted through the Data NoC. Red arrow: messages transmitted through the Control Noc.

With the availability of a PE in the cluster, the kernel migration starts. Actions represented in event **8** correspond to the kernel migration protocol (Section V-D): 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 $CM_{candidate}$. Next, the CM_F DMNI transfers the memory contents to $CM_{candidate}$ (**9**). Once the kernel received, the $CM_{candidate}$ 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.

VII. Results

The experiments are executed using a clock cycle accurate RTL SystemC model of the reference many-core platform (Figure 1). Applications and kernel are described in C language, compiled from C code and executed over cycle-accurate models of the processing cores.

The many-core contains 16 PEs, organized in 2x4 clusters. The experiments use two benchmarks to evaluate the recovery protocol: MPEG decoder (5 tasks) and Prod_Cons (2 tasks).

A. Proposed Protocol Overhead

This Section evaluates the overhead induced in the Application Execution Time (AET) when adopting the proposed recovery protocol, associated to a 64 KB kernel migration, with one task migration (10KB code and data). Figure 5 presents the steps for a task and a kernel migration. In Figure 5(a) $CM_{2,0}$ fails. The fault detection module notifies the fault by injecting in the control NoC router the *Fail CPU service*. The $CM_{0,0}$ knows that the $CM_{candidates}$ execute one task (task C). Thus, it is necessary a task migration before the kernel migration. In Figure 5(b) task C migrates from $SP_{2,1}$ to $SP_{1,1}$, in another cluster. When the task migration finishes, the kernel migrates to $SP_{2,1}$ (Figure 5(c)).

Each application executes with three different number of iterations. The MPEG executes 20, 40 and 60 iterations, and Prod_Cons executes 500, 700 and 900 iterations. Table I presents the execution time for both applications. The column baseline presents the AET without executing the proposed protocol. The last column $task + CM_F$ presents the AET with the protocol executing one task migration and the kernel migration. Note that the time to execute the protocol corresponds to **1.78 ms** (fourth column values minus the third column values). Hence, the AET overhead reduces when the number of executed interactions increases.

TABLE I
Application Execution Time (AET), in ms (@100MHz).

The percentage in the last column represents the overhead of the proposal w.r.t the baseline execution time.

	iterations	baseline	$task + CM_F$
MPEG	20 / 500	11.447	13.178 (15.1%)
+	40 / 700	17.824	19.607 (10.0%)
Prod_Cons	60 / 900	26.455	28.238 (6.73%)

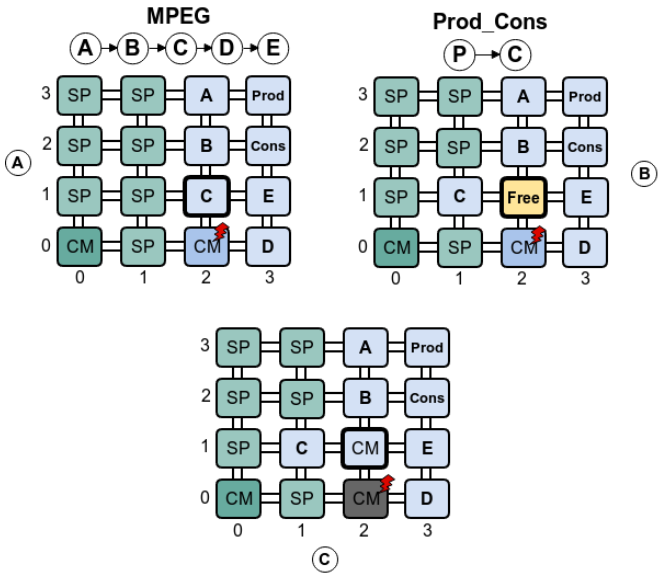

Fig. 5. MPEG and Prod_Cons task graphs, and applications mapping. Highlighted PE, $PE_{2,1}$, is the $CM_{candidates}$ to receive the kernel.

B. Proposed Protocol Steps Overhead

Table II details the time spent at each recovery protocol step. The first line presents the time when a fault was inserted and detected, 2.0 ms (Figure 5(a)). The second line shows the moment when task migration starts. The overhead induced by the task migration is 0.3 ms. The third line shows the moment when kernel migration starts. The overhead due to the kernel migration (64 KB) is in average 1.3 ms. For kernel sizes of 32 KB and 128 KB, the overhead is in average 0.7 and 2.6 ms, respectively.

TABLE II
PROTOCOL OVERHEAD FOR ONE AND TWO TASK MIGRATIONS.

	one task	two task
Fail CPU	2.000 ms	2.000 ms
Migration task	2.172 ms	2.130 ms
Wait Kernel	2.429 ms	2.633 ms
Unfreeze	3.777 ms	3.988 ms

VIII. 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 to 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.

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.

IX. ACKNOWLEDGEMENTS

Author Fernando Gehm Moraes is supported by FAPERGS (17/2551-196-1) and CNPq (302531/2016-5), Brazilian funding agencies.

REFERENCES

[1] G. Castilhos, M. Mandelli, G. Madalozzo, and F. G. Moraes, "Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes," in *ISVLSI*, 2013, pp. 153–158.

[2] R. Brillu, S. Pillement, F. Lemonnier, and P. Millet, "Cluster Based MPSoC Architecture: An On-chip Message Passing Implementation," *Design Automation for Embedded Systems*, vol. 17, no. 3-4, pp. 587–607, 2013.

[3] H. Kim, A. Vitkovskiy, P. V. Gratz, and V. Soteriou, "Use it or lose it: Wear-out and Lifetime in Future Chip Multiprocessors," in *MICRO*, 2013, pp. 136–147.

[4] O. Heron, J. Guilhemsang, N. Ventroux, and A. Giulieri, "Analysis of on-line self-testing policies for real-time embedded multiprocessors in DSM technologies," in *IOLTS*, 2010, pp. 49–55.

[5] B. Reddy, M. Vasantha, and Y. Kumar, "A Gracefully Degrading and Energy-Efficient Fault Tolerant NoC Using Spare Core," in *ISVLSI*, 2016, pp. 146–151.

[6] N. Dutt, A. Jantsch, and S. Sarma, "Self-Aware Cyber-Physical Systems-on-Chip," in *ICCAD*, 2015, pp. 46–50.

[7] H. Tajik, B. Donyanavard, N. Dutt, J. Jahn, and J. Henkel, "SPM-Pool: Runtime SPM Management for Memory-Intensive Applications in Embedded Many-Cores," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 1, pp. 25:1–25:27, 2016.

[8] F. Barreto, A. M. Amory, and F. G. Moraes, "Fault Recovery Protocol for Distributed Memory MPSoCs," in *ISCAS*, 2015, pp. 421–424.

[9] L. Caimi, V. Fochi, E. Wachter, D. Munhoz, and F. G. Moraes, "Secure Admission and Execution of Applications in Many-core Systems," in *SBCCI*, 2017, pp. 65–71.

[10] J. Paul *et al.*, "Self-adaptive Corner Detection on MPSoC Through Resource-aware Programming," *Journal of System Architecture*, vol. 61, no. 10, pp. 520–530, 2015.

[11] Y. Y. Chen *et al.*, "Path-Diversity-Aware Fault-Tolerant Routing Algorithm for Network-on-Chip Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 838–849, 2017.

[12] A. Kamran and Z. Navabi, "Stochastic Testing of Processing Cores in a Many-core Architecture," *Integration, the VLSI Journal*, vol. 55, pp. 183–193, 2016.

[13] B. Bhowmik, J. K. Deka, S. Biswas, and B. B. Bhattacharya, "On-line Detection and Diagnosis of Stuck-at Faults in Channels of NoC-based systems," in *SMC*, 2016, pp. 4567–4572.

[14] V. Tsoutsouras, D. Masouros, S. Xydis, and D. Soudris, "SoftRM: Self-Organized Fault-Tolerant Resource Management for Failure Detection and Recovery in NoC Based Many-Cores," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 144:1–144:19, 2017.

[15] E. A. Carara, R. P. de Oliveira, N. L. V. Calazans, and F. G. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," in *ISCAS*, 2009, pp. 1345–1348.

[16] E. Wachter, L. L. Caimi, V. Fochi, D. Munhoz, and F. G. Moraes, "BrNoC : A broadcast NoC for control messages in many-core systems," *Microelectronics Journal*, vol. 68, pp. 69–77, 2017.

[17] M. Ruaro, F. B. Lazzarotto, C. A. Marcon, and F. G. Moraes, "DMNI: A specialized network interface for NoC-based MPSoCs," in *ISCAS*, 2016, pp. 1202–1205.

[18] P. Meloni *et al.*, "System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach," in *DSD*, 2012, pp. 517–524.

[19] D. Wentzlaff *et al.*, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[20] V. Fochi, E. Wächter, A. Erichsen, A. M. Amory, and F. G. Moraes, "An integrated method for implementing online fault detection in NoC-based MPSoCs," in *ISCAS*, 2015, pp. 1562–1565.