System Management Recovery Protocol for MPSoCs

Vinicius Fochi, Luciano L. Caimi, Marcelo Ruaro, Eduardo Wachter, Fernando G. Moraes
FACIN - PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
{vinicius.fochi, luciano.caimi, marcelo.ruaro, eduardo.wachter}@acad.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—The advances in silicon technology lead to systems with hundreds of processors, the NoC-based MPSoCs. However, the higher fault probability in deep sub-micron technologies shortens the integrated circuits lifetime. Operating systems enable to execute distributed applications in the MPSoC processing elements (PEs). Large systems require PEs dedicated to management purposes, for example, execute the task mapping, handle monitoring data, and run self-awareness adaptation. This paper addresses an MPSoC hierarchically organized: PEs with an embedded operating system executing the applications (S_{PE}) and dedicated PEs manage at runtime the system resources (M_{PE}). A rich literature presents fault-tolerant proposals for the hardware and software components of the MPSoC, but there is a significant gap related to fault-tolerant approaches at the system level, i.e., related to the PEs with the function to manage the system. Consider for example an M_{PE} responsible for managing a set of S_{PE}s. A fault in an M_{PE} prevents the access to the set of S_{PE}s to execute new applications. The goal of this paper is to present a method to determine when an M_{PE} became faulty, and propose a protocol to migrate the management software safely to an S_{PE}. The management data is preserved, without saving the context in redundant structures. The proposal is transparent to the applications executing in the system, with a small execution overhead observed during the management migration, presented in the results Section.**

***Index Terms*—MPSoC; NoC; System Management; Fault-recovery protocol; Fault-tolerance.**

I. Introduction

The continuous development of large multiprocessor systems-on-chip (MPSoCs) resulted in systems with dozens of embedded processors. Networks-on-chip (NoCs) provide enhanced performance and scalability for communication. Large systems require processing elements (PEs) dedicated to *management purposes*, for example, execute the task mapping, handle monitoring data obtained from sensors and estimation functions, and run self-awareness adaptation (e.g. quality-of-service, DVFS control, aging, temperature) [1] [2].

MPSoC with a hierarchical organization ensure scalability at the management level. This organization assigns distinct roles to PEs: cluster managers (*CM*) and slave processors (*SP*) [3]. With such organization, the MPSoC contains virtual regions, named *clusters*, with one *CM* and a set *SPs* per cluster. A cluster may increase its size at runtime, borrowing *SPs* from neighbor clusters, in a process named reclustering.

Transistors, vias, and wires degrade faster over time in deep sub-micron technologies, causing transient faults and permanent errors, thus shortening integrated circuits lifetime [4]. For these reasons, reliability becomes a key issue in MPSoC architecture design [5]. Classical fault-tolerant approaches, as TMR or spare components [6], do not comply with today's requirements of silicon area and power dissipation. Due to the way it is built, an MPSoC provides a set of replicated structures (PEs), where a healthy component can execute the faulty component functions, without penalizing the system performance.

It is worth to differentiate the consequences of a permanent fault in *SPs* and *CMs*. A fault in a PE executing a user application (*SP*) compromises the application, being possible to remap the application [7]. The effect of a fault in a *CM* is more severe than a fault in an *SP*, because it may halt the entire cluster, making the set of *SPs* controlled by the faulty *CM* unavailable.

The *goal* of the paper is to present a protocol to detect at runtime faulty *CMs* and a recovery method to migrate the functions of the faulty component to a healthy one. The protocol detects faulty *CMs*, starting the process to delegate this manager capability to another PE. Briefly, the method detects a faulty *CM*, choose an *SP* to became the new *CM*, freeze the tasks owned by the failed *CM*, migrate the memory contents to the new *CM*, reboot the *CM*, and unfreeze the tasks without restarting them.

The main original *contribution* of the proposal is a runtime fault recovery protocol targeting the management of the MPSoC, with the following features: (*i*) migration of the management functions to a new PE; (*ii*) the management context is preserved without saving the context in redundant structures. Related works focus on specific components of the MPSoC, as the processor or the NoC, without a systemic view of the system.

This paper is organized as follows. Section II presents related work. Section III details the system architecture. Section IV details the fault model. Section V presents the protocol for fault handling in manager processors. Section VI presents the results, and Section VI concludes this paper.

II. Related Work

Azad et al. [8] propose a system with a set of components for detection, classification and recovering from faults. The system contains a System Health Monitoring Unit (SHMU). The SHMU provides a holistic view of the system health status, using a memory which keeps different mapping and scheduling solutions based on the current fault configuration of the system. The system has online checkers for fault detection,

978-1-5386-4034-0/17/$31.00 ©2017 IEEE

able to capture faults with low detection latency and providing the fault information for SHMU. The SHMU can deal with faults on PEs, routers, and links. The main drawback of the approach is the centralized SHMU, compromising scalability, and the system model. The Authors model the system with Phyton, being not possible to infer the behavior of the system in the presence of faults.

Walters et al. [9] describe fault-tolerant strategies for the Maestro many-core processor. The Authors adopt a software-based fault tolerance, memory controllers, PLR (process-level replication), TLR (thread-level replication), kernel level checkpoint/rollback, distributed heartbeat implementation, and hybrid or application-specific fault-tolerant strategies. The PLR creates two redundant processes for a single application, with the same application running on different tiles. The output files are compared at the end of execution, and if they do not match, a majority rule determines the correct output. The TLR is similar to PLR, but used for thread-level, providing a fine-grain control of the replication process. Heartbeat is a method used to determine if a processor continues its execution or not. A shared memory keeps the timestamps of each processor. A timeout mechanism determines when the processor is considered failed.

Bolchini et al. [10] describe a system with an adaptive level of reliability. The work presents a fault management layer at the operating system level. This layer has a strategy for dynamically adapting the reliability at run-time. The fault management layer contains three methods: duplication with comparison (DWC), triplication (TMR), duplication with comparison and re-execution (DWCR). The DWC create an application replica and compare the outputs. The TMR creates two replicas of the original application and compares the outputs. The DWCR creates an application replica and compares the results. If an error is detected, a third task replica is created and executed. The method has an observe-decide-act (ODA) control loop. The observe module collect the status and execution data from the system. The decide module measure the data collected and analysis the system according to the specified metrics and goals. The act module can modify the system to alter its behavior, as activate or deactivate a PE.

Previous works present fault-tolerant methods focusing on the applications' execution, using methods well established in distributed systems: software replication. 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 PE (*CM*) to a healthy PE.

III. System Architecture

Figure 1 presents the modules of the reference many-core platform adopted in the present work. The architecture contains a set of PEs interconnected by a *data NoC* and a *control NoC*. One *CM* has an interface with the external environment to the MPSoC 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), a dual-port private memory, the data and control routers. The hardware of the PEs is the same, being the role assigned to the PEs made by software: *SPs* executes users' tasks, supporting multitasking; *CMs* manage a given cluster.

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.

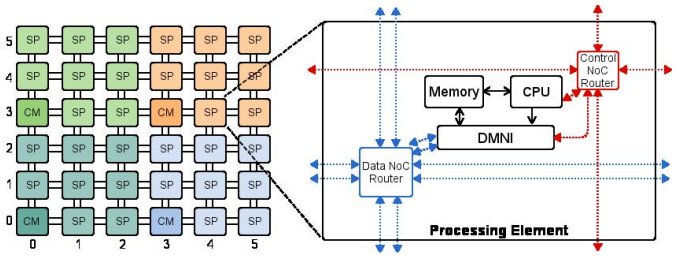

Fig. 1: A 6x6 instance of the reference many-core system with 3x3 clusters.

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* transfers the control messages. Control messages have different constraints from data messages. Control messages have to reach their target(s) as fast as possible, and can not be exposed to congestion effects (e.g. fault notification). Control messages may have more than one target (e.g. freeze all tasks of a given application or cache protocols). Also, control messages are much less frequent than application messages, with a small payload. The *control NoC* has the following features (similar architecture to [11]): adoption of broadcast as the default transmission mode, bufferless router, each message with one flit. The current work uses the *control NoC* for the following purposes:

- monitoring the status of *CMs*;
- freeze an application(s) when a *CM* fails;
- notify an *SP* that it will become a new *CM*;
- notify a DMNI module to transfer the memory contents of a PE to a new system address system;
- notity *SPs* with the new *CM* address;
- unfreeze application(s) after the *CM* migration.

An important feature of the architecture 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 of the memory contents when the processor has a permanent fault, is commonly adopted in fault tolerant approaches [12].

The method herein presented may be applied to homogeneous or heterogeneous MPSoCs. Summarizing, the method

presented in Section V 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 [13]; (*iii*) a memory module that can be read/write directly by the network interface [12].

IV. Fault Model

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

- *Healthy modules of the PE*: 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 [14], and according to the severity of the transient faults trigger the proposed protocol.
- *Faulty module of the PE*: CPU. The proposed method is fired a when a permanent fault is detected in a manager PE.

The basis of the fault recovery method is a monitoring process between *CMs*. Each *CM* receives messages periodically from a neighbor *CM*. When a *CM* has a permanent fault, the recovery process starts. The goal of the recovery process is to transfer the memory contents of the faulty *CM* to a healthy PE. The DMNI of the faulty *CM* manages this process.

A. Fault Detection Mechanisms

This section presents examples of fault detection methods, that can be applied to the PE modules, and used by be current work. All techniques cited bellow can initiate the proposed protocol. 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. The Madness project [12] adopts two approaches to detect faulty processors: self-testing using a pre-computed signature for non-critical applications, or N-modular redundancy at the software level. Using these methods, the Authors present a system level adaptive and fault-tolerant techniques to reduce the performance loss by using dynamic remapping (task migration) of faulty PEs. In [15] the Authors propose a Runtime Module Configuration with a 3-mode configurable encoder. The goal is to change the encoder mode according to the number of faults occurring at the NoC links. The method encodes packets and optimizes the fault coverage of the NoC.

Fault detection at the processor level. In [16] a general purpose device (GPD) creates a test pattern, sending it to the processor. The processor applies the test pattern and sends the results back to the GPD. Faulty tiles are bypassed and replaced by another processor via an embedded resource manager implemented in software. The Authors in [9] adopt a software-based fault tolerance, PLR (process-level replication), TLR (thread-level replication), kernel level checkpoint/rollback and distributed heartbeat implementation.

Fault detection at the router level. The proposal in [17] inserts multiplexers at the input ports to enable port swapping, and a bypass bus enables to connect input ports to output ports when the internal crossbar fails. Zhang et al. in [18] present a dual-input crossbar design targeting performance and power reduction. The crossbar duplication enables fault tolerance at the router level. When a crossbar failure is detected, all the inputs ports are forwarded to another crossbar. The proposal in [19] focuses on transient errors in the router using ECC to prevent packet loss, incorrect routing, and network congestion. An error detection module request re-computation if a fault is detected. It also includes an error correction module after the crossbar to prevent error propagation.

Fault detection at the link level. In [20] the Authors propose a fault-tolerant method with a gracefully degrading link-level, proportional to the number of faults detected in the link. In [21] the Authors implements an error recovery technique for NoCs with the goal to protect network links against crosstalk effects using CRC modules.

V. Protocol for Fault Handling in Manager Processors

This Section presents the fault recovery protocol, including the system monitoring method to detect faulty *CMs* (V-B), the criteria to select the address of a new *CM* (V-C), and the protocol to migrate the *CM* memory contents (V-D). The following definitions of terms are adopted:

Definition 1. $Ward\ message$: a monitoring message transmitted through the *control NoC* between *CMs*.

Definition 2. $Ward\ pair$: a pair of *CMs* responsible for supervising each other, by exchanging periodically ward messages.

Definition 3. CM_H: healthy CM responsible to execute the management recovery protocol.

Definition 4. CM_F: faulty CM stopped due to a permanent fault in the processor.

Definition 5. $CM_{candidate}$: PE that will assume the role of the CM_F.

A. Definition of manager pairs

The proposal starts by defining the *ward pairs*. Each *CM* selects its pair at runtime, at system startup. Note that the definition of the *ward pairs* is logical, not physical. At system startup the *ward pairs* are physically aligned, but after migrating a *CM* to a new position this arrangement changes. For this reason, the monitoring process uses broadcast (*control NoC*) to exchange messaged between *ward pairs*.

The method defines initially horizontal *ward pairs*. Consider Figure 2 as an example. The horizontal *ward pairs* are $\{CM_{0,0}, CM_{3,0}\}$, $\{CM_{0,3}, CM_{3,3}\}$ and $\{CM_{0,6}, CM_{3,6}\}$. If the number of *CM* columns is odd, the second step defines vertical *ward pairs* at the rightmost *CM* coordinate. In this example, one vertical *ward pair* is created, $\{CM_{6,6}, CM_{6,3}\}$. Finally, as $CM_{6,0}$ has no pair to supervise, its ward is the last *CM* address, $CM_{6,3}$.

Thus, $CM_{6,3}$ is in charge to monitor the status of $CM_{6,0}$ and $CM_{6,6}$. Note that $CM_{6,0}$ sends periodically `Ward_Msgs`, and its supervision function responsible to trigger the management migration is disabled.

$$\begin{array}{ccccc} CM_{0,6} & \longleftrightarrow & CM_{3,6} & & CM_{6,6} \\ & & & & \updownarrow \\ CM_{0,3} & \longleftrightarrow & CM_{3,3} & & CM_{6,3} \\ & & & & \uparrow \\ CM_{0,0} & \longleftrightarrow & CM_{3,0} & & CM_{6,0} \end{array}$$

Fig. 2: Example of ward pairs definition.

B. Ward Messages

After defining the *ward pairs*, each *CM* exchange periodically `Ward_Msgs`. The definition of the interval between these messages is a design-time parameter. Larger periods delays the time to recover from a faulty *CM*, while shorter periods may lead to false positives. The false positives may occur if the *CM* is executing management functions, delaying the answer to its pair, which will consider it faulty, starting the recovering process.

In the present work, the monitoring messages are sent every millisecond. Every millisecond each *CM* sends a `Ward_Msg` through the *control NoC* to its pair. The message is sent by broadcast and only the destination *CM* handles this message. If a *CM* sends three `Ward_Msgs` without receiving any answer from its pair, the *CM* that did not reply is considered faulty, starting the recovering process. The `Ward_Msg` uses the control NoC and broadcast transmissions because the *CM* address changes when the recovering process occurs, misaligning the *CM* addresses. According to the requirements of the platform, it is possible to parametrize the interval between the `Ward_Msgs`. Smaller intervals between `Ward_Msgs` leads to higher execution overhead time while higher intervals increase the recovery time.

The detection of a permanent fault in the processor induces its isolation by test wrappers. Thus, all messages transmitted before the detection of the faulty processor are discarded, and the source of the message retransmits it later (every message transmitted to a *CM* requires an acknowledgment).

C. Definition of the PE to Receive the CM Memory

If a fault disables a *CM* processor, the memory contents of this processor must migrate to a new PE. The process to define a PE to receive the memory contents occurs as follows. At system startup, the closest PE to *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 PE with the minimum number of tasks assigned to it. If the $CM_{candidate}$ has tasks assigned to it, all tasks executing in this PE must be migrated to enable the reception of the *CM* memory contents.

Thus, after any application mapping, the *CM* computes the $CM_{candidate}$ address and transmits it to its *Ward pair*. If necessary, the identification of the tasks to migrate is also transmitted.

D. Management Recovery Protocol

Figure 3 presents the management recovery protocol, assuming: *CM1* as CM_F, *CM0* as CM_H, *SP7* as $CM_{candidate}$.

CM0 and *CM1* are a *CM* pair, exchanging periodically `Ward_Msgs` (event **1** at Figure 3). At a given moment (event **2**), *CM1* becomes unresponsive and after three unanswered `Ward_Msgs` it is considered faulty. Then, its *CM* pair starts the process to promote *SP7* to *CM*. The first action is to inject a `Freeze` message to all PEs (event **3**). Only tasks managed by CM_F stops the execution. Note that as this message is sent through the control NoC (in broadcast), tasks executing in another clusters due to the reclustering process are also freezed.

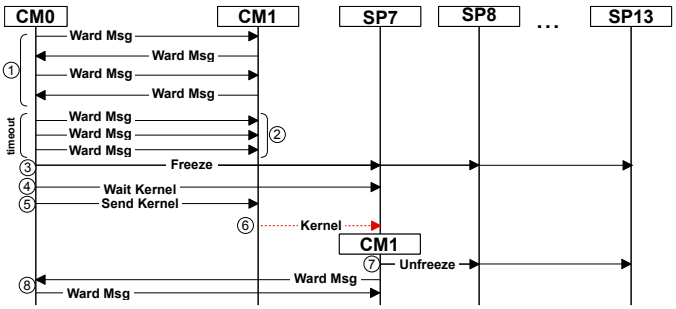

Fig. 3: Management Recovery Protocol. Continuous lines: messages exchanged by the *control NoC*. Dotted red line (event 6): message exchanged by the *data NoC*.

An SP receiving a `Freeze` message verifies if exists in its TCB (Task Control Block) a task managed by CM_F. The `Freeze` message does not stop the task immediately. To avoid messages losses, the task must be in a *safe* state. A *safe* state if defined as: the task to freeze is 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*. Any task managed by CM_F goes to a *freeze state*, avoiding its scheduling by the kernel.

Next, *CM0* notifies *SP7* that it will receive the kernel executing in *CM1* (event **4**), `Wait Kernel` message. The SP defined as $CM_{candidate}$ handles this message. A special field in the packet header of this message defines that the DMNI module process the payload, not the processor. The action executed by this message is twofold: hold the processor and configure to DMNI module to write incoming packets in the memory from address zero.

Figure 4 details the management recovery protocol events on the faulty *CM* (*CM1*). The DMNI module handles the kernel migration in this situation.

After notifying the $CM_{candidate}$, *CM0* (CM_H) sends a message to the *CM1* (CM_F): `Send Kernel` (event **5** - Figures 3 and 4). The DMNI handles this message, transferring the memory contents (code and data) to *SP7* ($CM_{candidate}$), using the data NoC. After transferring the memory contents, the DMNI is programmed to avoid any transmission from the faulty processor, preventing Bizantine faults. Note that this is the only message transferred using the data NoC in the protocol. All other messages use the control NoC.

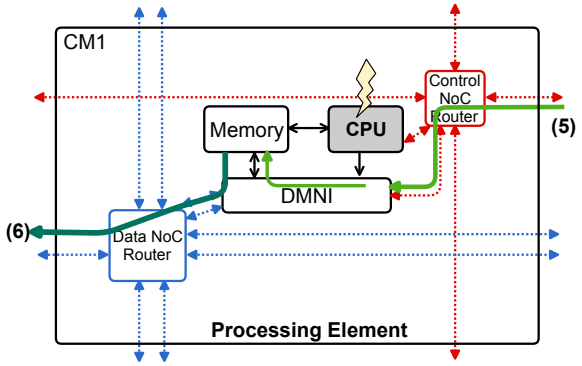

Fig. 4: Events 5 and 6 of the protocol, on *CM1* (faulty *CM*).

The *SP7* ($CM_{candidate}$) after received the kernel (event **6**), restart the execution, now as a *CM*. Once the migration process finishes at the $CM_{candidate}$, the DMNI returns to the normal operation, and the CPU is restarted. As the kernel migration transferred the data memory, the *CM* data structures are preserved. Thus, the restart of the new *CM* updates few data structures, such as its network address. After restarting, the new *CM* sends an `Unfreeze` message to the stopped task (event **7**). This message unfreezes the tasks managed by the new *CM* and also transmits the *CM* address to the *SPs*. The new *CM* restarts the `Ward_Msgs` with its pair (event **8**).

VI. Results

The experiments are executed using a clock cycle accurate RTL SystemC model of the reference MPSoC 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 MPSoC contains 16 PEs, organized in 2x4 clusters. The experiments use two benchmarks to evaluate the approach (Figure 5): MPEG decoder (5 tasks) and DTW (6 tasks).

A. Overhead of the Proposed Protocol

This Section evaluates two scenarios. The first one evaluates the overhead due to the `Ward_Msg` messages. The second one evaluates the protocol overhead to freeze tasks and migrate the kernel to the new *CM*. The two scenarios execute according to the mapping illustrated in Figure 5.

Each application executes with three different number of iterations. The MPEG executes 20, 40 and 60 iterations, and the DTW executes 160, 240 and 320 iterations. Table I presents the execution time for both applications. The column *no ward*

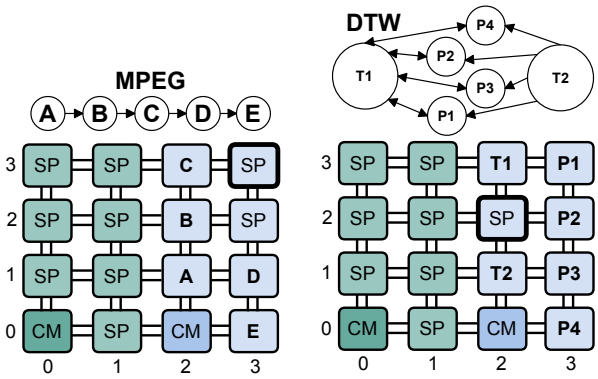

Fig. 5: MPEG and DTW task graphs, and applications mapping. Highlighted PEs, $PE_{3,3}$ and $PE_{2,2}$, are the $CM_{candidates}$ to receive the kernel.

presents the Application Execution Time (AET) without the proposed protocol. The column *with ward* presents the AET with the proposed protocol, but without a faulty *CM*. The last column, $ward + CM_F$ presents the AET when the protocol is executed, migrating CM_F to $CM_{candidate}$: $CM_{2,0}$ to $CM_{3,3}$ and $CM_{2,0}$ to $CM_{2,2}$, for the MPEG and DTW scenarios respectively.

TABLE I: Application Execution Time, in ms. The percentage in the last column represents the overhead of the proposal w.r.t the baseline execution time (*no ward*).

	iterations	*no ward*	*with ward*	$ward + CM_F$
MPEG	20	9.246	9.256	10.068 (8.89%)
	40	17.994	18.005	18.803 (4.50%)
	60	26.737	26.749	27.584 (3.17%)
DTW	160	14.315	14.326	15.173 (5.99%)
	240	21.231	21.241	22.170 (4.42%)
	320	28.146	28.156	29.132 (3.50%)

The overhead induced the `Ward` protocol is 0.01 ms, regardless the number of iterations executed by the applications. This time is constant because it corresponds to the interference of the `ward` messages with the application mapping, which occurs once (both actions, monitoring and mapping run in the *CM*).

The overhead induced by the migration of the *CM* memory contents is in average 0.877 ms (from 0.809 ms to 0.986 ms). This result also has a small variation because only 1 migration occurred. As expected, the overhead reduces according to the number of executed iterations (percentage values in the last Table column). *This result shows the effectiveness of the proposal, with a small impact of the protocol in the application execution time*. Note that the delay to migrate the *CM* is proportional to the memory size. The memory size for these experiments is 64 KB. The overhead is in average 0.4 ms and 1.6 ms for a memory size of 32 KB and 128 KB, respectively.

B. Execution Overhead in Applications

This Section presents the latency to execute one iteration of each application of Figure 5. The graphs in Figure 6 represents

in the x-axis the iteration number and in the y-axis the number of clock cycles to execute one iteration. As there are no disturbing applications in these scenarios, the iteration latency is constant for both applications. When the *CM* migrates, the application is stopped, increasing the latency of *one* iteration only. This result is in agreement with the previous results, showing that the overhead of the proposal is small, and the effect on the applications is negligible. Take for example the MPEG application. According to the latency graph, only one decoded frame is delayed, which is in practice imperceptible by the users.

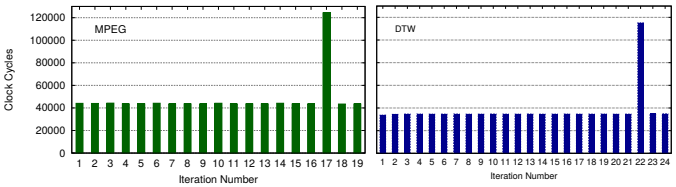

Fig. 6: MPEG and DTW latency to execute one iteration of the applications.

VII. Conclusion

This work presented a runtime protocol for management recovery in NoC-based MPSoCs. The proposal includes a monitoring method that determines when a manager processor fails, the `Ward` protocol, and a method to safely migrate the management software to a new processing element, assuming a protected memory. The results displayed a small overhead to add the `Ward` protocol, as weel 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 single point of failure, i.e., enable multiple *CMs* to receive application requests.

VIII. Acknowledgement

Authors Eduardo Wachter and Fernando Gehm Moraes are supported by CNPq funding agency.

References

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

[2] 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, Oct. 2016.

[3] 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.

[4] 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.

[5] 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.

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

[7] F. F. S. Barreto, A. M. Amory, and F. G. Moraes, "Fault recovery protocol for distributed memory MPSoCs," in *ISCAS*, 2015, pp. 421–424.

[8] S. P. Azad, B. Niazmand, J. Raik, G. Jervan, and T. Hollstein, "Holistic Approach for Fault-Tolerant Network-on-Chip based Many-Core Systems," *CoRR*, vol. abs/1601.07089, 2016. [Online]. Available: http://arxiv.org/abs/1601.07089

[9] J. P. Walters, R. Kost, K. Singh, J. Suh, and S. P. Crago, "Software-based fault tolerance for the Maestro many-core processor," in *IEEE Aerospace Conference*, 2011, pp. 1–12.

[10] C. Bolchini, M. Carminati, and A. Miele, "Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems," *Journal of Electronic Testing: Theory and Applications*, vol. 29, no. 2, pp. 159–175, Apr. 2013.

[11] E. Wachter, A. Erichsen, A. Amory, and F. G. Moraes, "Topology-Agnostic fault-tolerant NoC routing method," in *DATE*, 2013, pp. 1595–1600.

[12] P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin, and M. Sami, "System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach," in *DSD*, 2012, pp. 517–524.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sept 2007.

[14] 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.

[15] T. Boraten and A. Kodi, "Runtime Techniques to Mitigate Soft Errors in Network-on-Chip (NoC) Architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[16] T. D. T. Braak, S. T. Burgess, H. Hurskainen, H. G. Kerkhoff, B. Vermeulen, and X. Zhang, "On-line dependability enhancement of multiprocessor SoCs by resource management," in *SoC*, 2010, pp. 103–110.

[17] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, "Vicis: A reliable network for unreliable silicon," in *DAC*, July 2009, pp. 812–817.

[18] Y. Zhang, R. Morris, D. DiTomaso, and A. Kodi, "Energy-Efficient and Fault-Tolerant Unified Buffer and Bufferless Crossbar Architecture for NoCs," in *IPDPS*, 2012, pp. 972–981.

[19] Q. Yu, M. Zhang, and P. Ampadu, "Exploiting inherent information redundancy to manage transient errors in NoC routing arbitration," in *NoCS*, 2011, pp. 105–112.

[20] A. Vitkovskiy, V. Soteriou, and C. Nicopoulos, "A Dynamically Adjusting Gracefully Degrading Link-Level Fault-Tolerant Mechanism for NoCs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 8, pp. 1235–1248, 2012.

[21] A. H. Lucas and F. G. Moraes, "Crosstalk fault tolerant NoC - Design and evaluation," in *VLSI-SoC*, 2009, pp. 115–120.