2012 15th Euromicro Conference on Digital System Design

System Adaptivity and Fault-tolerance in NoC-based MPSoCs: the MADNESS Project Approach

Paolo Meloni*, Giuseppe Tuveri*, Luigi Raffo*, Emanuele Cannella†, Todor Stefanov†, Onur Derin‡,
Leandro Fiorin‡ and Mariagiovanna Sami‡
*Department of Electrical and Electronic Engineering
University of Cagliari, 09123 Cagliari, Italy
Email: {paolo.meloni, giuseppe.tuveri, luigi}@diee.unica.it
†LIACS, Leiden University, 2333 CA Leiden, The Netherlands
Email: {cannella, stefanov}@liacs.nl
‡ALaRI, Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland
Email: {derino, leandro.fiorin}@liacs.nl, sami@elet.polimi.it

***Abstract*—Modern embedded systems increasingly require adaptive run-time management. The system may adapt the mapping of the applications in order to accommodate the current workload conditions, to balance load for efficient resource utilization, to meet quality of service agreements, to avoid thermal hot-spots and to reduce power consumption. As the possibility of experiencing run-time faults becomes increasingly relevant with deep-sub-micron technology nodes, in the scope of the MADNESS project, we focus particularly on the problem of graceful degradation by dynamic remapping in presence of run-time faults.**

In this paper, we summarize the major results achieved in the MADNESS project until now regarding the system adaptivity and fault tolerant processing. We report the first results of the integration between platform level and middleware level support for adaptivity and fault tolerance. A case study demonstrates the survival ability of the system via a low-overhead process migration mechanism and a near-optimal online remapping heuristic.

I. Introduction

Designing multi-processor embedded systems, effectively exploiting the integration capabilities provided by modern technology processes and, at the same time, complying with the complexity of current and future applications, is a complicated activity. IP cores provided by different parties have to be efficiently integrated and programmed exposing the designer to a wide range of degrees of freedom. Moreover, the landscape of applications on the market pushes the needs for high performance and, nevertheless, requires the underlying architecture to be flexible. The MADNESS project aims at the definition of novel methods supporting the designer to accomplish the previously mentioned complex objectives. The work envisioned within the project will result in the construction of an integrated framework for the application-driven design of MPSoCs. The framework, whose overview is presented in detail in [1], will be composed of several tools and IPs, interacting to achieve the identification and the implementation of the optimal system configuration. A relevant part of the project is focused on Design Space Exploration (DSE), which is supported with advanced FPGA-based prototyping techniques. The DSE process is applied and evaluated on industrially-relevant design cases which involve the integration of industrial components, such as application-specific processing elements and structured interconnects. With respect to similar projects, one main point of novelty in MADNESS is the emphasis on runtime system adaptivity and fault tolerance as two main factors to be considered when designing the system.

In this paper, we summarize the major results achieved in the MADNESS project until now regarding the system adaptivity and fault tolerance mechanisms. We report the first results of the integration between platform level support and middleware level support for adaptivity and fault tolerance. A case study demonstrates the survival ability of the system in reaction to faults in the processing elements by online remapping mechanisms of the application tasks.

II. The MADNESS Project approach

Modern embedded systems increasingly require adaptive run-time management. The workload that a system has to deal with cannot be completely predicted at design-time. For example, new applications can be loaded at run-time or, to comply with limited power and energy budget, power-aware application management techniques are often needed, such as the dynamic balancing of the workload between the processing cores. Moreover, with deep-sub-micron technology, the possibility of experiencing faults in the circuitry is significant, requiring the system to feature support for graceful degradation of the performance in case of malfunctioning.

The research leading to these results has received funding from the European Community Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 248424, MADNESS Project, from ARTEMIS JU - ASAM Project, and by the Region of Sardinia, Young Researchers Grant, PO Sardegna FSE 2007-2013, L.R.7/2007 “Promotion of the scientific research and technological innovation in Sardinia”.

978-0-7695-4798-5/12 $26.00 © 2012 IEEE
DOI 10.1109/DSD.2012.122

CPS Conference Publishing Services

Within MADNESS, to cope with these issues, we have devised techniques that allow to change the mapping of the application processes onto the processing cores at run-time. The development of these techniques required the introduction of dedicated support at several levels.

At the architectural level, the MADNESS approach considers a distributed-memory tile-based template, where tiles are interconnected through a NoC, to support the high flexibility and scalability demands. The architectural template is customizable in terms of the number of processors and network topology. It has been extended with newly developed hardware IPs that facilitate the run-time management and that expose to the applications the needed communication and synchronization primitives. Extensions will be described more in detail in Section IV.

At the software level, a specific layered infrastructure has been devised, that enables the execution of applications described using the Polyhedral Process Network (PPN) model of computation [2]. PPNs were chosen because their simple operational semantics allows low-overhead management of the state of the application tasks at run-time. The software infrastructure, actually in charge of managing the mapping of the PPN tasks, the communication between them and the migration process, will be described in Section V.

Moreover, fault tolerance support has been introduced at both software and hardware levels. The idea is to improve dependability of the system by exploiting the migration method in case of run-time faults in the processing cores. The tasks mapped on faulty cores have to be migrated to fault-free ones at run-time, so that the application can continue its execution without disruption. To this aim, several extensions to the migration mechanism are needed. Firstly, fault detection must be enabled so that the migration can be triggered. Secondly, given that a faulty processor cannot participate in the remapping process, dedicated hardware is needed to ensure the migration functionality to survive in case of malfunctioning. Finally, a remapping decision must be taken in such a way to incur the least performance degradation. The details of the proposed solutions are described in Section VI.

III. Related work

A survey regarding the state-of-the-art in run-time management is provided in [3], where system adaptivity and fault tolerance are envisioned as important research challenges. The infrastructure developed in our work addresses system adaptivity and fault tolerance by allowing process remapping at run-time. In addition, our work includes a set of heuristics that can make remapping decisions in case of faults.

In [4], Almeida et al. describe a framework oriented to system adaptivity which is close to our approach. In their work, the goals of scalability and system adaptivity are achieved using a completely distributed task migration policy over a purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network model of computation. However, our work is fundamentally different because it enables the migration to happen at any time within the main body of the processes. This is a basic requirement in order to allow fault tolerance, because faults can happen at any time. By contrast, in [4] the process migration is enabled only at fixed points during the execution of processes.

Dynamic task remapping is also performed in [5], [6] by means of a task migration mechanism implemented at user-level or middleware/OS level respectively. Both these approaches require the user to specify checkpoints in the code at which migration can take place. In our approach this is not needed because the state that has to be migrated is automatically determined, thanks to the properties of the adopted model of computation (Polyhedral Process Networks [2]). Another difference concerns the inter-processor communication implementation. The systems considered in [5], [6] use a shared memory paradigm to implement inter-processor communication. We argue that our approach, which uses a pure distributed memory, intrinsically provides better scalability.

Task remapping for reliability purposes has been investigated in [7] with the goal of throughput minimization on multicore embedded systems. The fundamental difference from our approach is the use of design-time analysis for all possible scenarios and the storage of all remapping information in the memory. We argue that this technique incurs a large memory requirement to store all fault scenarios.

In [8], a system-level fault-tolerance technique for application mapping, which aims at optimizing the entire system performance and communication energy consumption, is proposed. In particular, the authors address the problem of spare core placement and its impact on system fault-tolerance properties, and propose a run-time fault-aware technique for allocating the application tasks to the available, reachable, and fault-free cores of embedded NoC platforms. In [8], application components running on a faulty core are migrated altogether to available non-employed spare cores, whereas, in our approach, tasks on the faulty core can possibly be remapped to different fault-free cores.

IV. Architectural support

As previously mentioned, in the proposed approach the system architecture can be seen as a network of tiles, interconnected by means of an NoC communication infrastructure, as depicted in Fig. 1.

The communication network is built by using an extended version of the the ×pipes-lite library of synthesizable components [9]. The topology can be completely arbitrary, since it includes a fabric of routers and links that can be almost entirely customized. Network access points are Network Interfaces (NI), that are in charge of constructing the packets on the basis of the communication transactions requested by the cores. NIs, placed at the interface between processing elements and the communication network, have been extended with support for message-passing communication model. A programmable message manager with DMA capabilities is integrated with the NI inside a module called Network Adapter (NA), described more in detail in Section IV-B. The processing element architecture is not fixed. Any kind of RISC or ASIP processor with standard bus-based signal interface can be easily integrated. No instruction set extensions are needed, since communication and synchronization mechanisms are

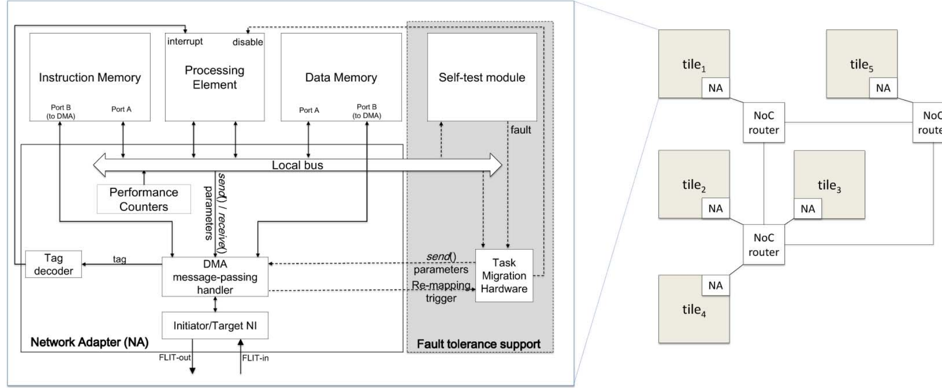

Fig. 1. A general overview of an example template instance

managed accessing memory-mapped registers at the network interfaces. The template obvioulsy allows the connection of peripheral controllers that can be connected as network nodes and receive transactions initiated by processing elements.

A. Programming model

Reference primitives implementing message-passing communication are built, according to the general definition of such model, upon two base functions: *send()* and *receive()*. These two primitives are implemented in C, and interact with the hardware structures described in Section IV-B. According to the usual message-passing signatures, to send a message with a *send()*, the programmer has to specify the address (*SendAddress* hereafter) inside the private memory that contains the information to be sent (message data), a tag assigned to the message (*SendTag*), the size of the transfer (*SendDim*), and the ID of the destination processor (or process, in case of multi-context execution in the processing elements - *SendID*). The *receive()* parameters are the tag of the expected message (*ReceiveTag*), the sender ID (*ReceiveID*) and the address where the received message data has to be stored (*ReceiveAddress*). Two implementations of the *receive()* are provided, with blocking and non-blocking behaviour.

B. Message Passing support

The *Network Adapter* architecture is depicted in Fig. 1 (left side). Both the instruction and data private memories of the processor have two access ports, in order to allow the processor to keep on accessing code and data from one instruction and one data port, while, at the same time, the other ports can be used to directly load/store data from/to the memory in case of message send/receive. In this way, communication and computation can overlap, potentially leading to a significant speed-up. The *NA* integrates a local bus, that, according to the address requested by the processor interface, enables access to:

- the private memory,
- a module called *DMA message-passing handler* (MPH),
- a set of performance counters to obtain statistics about the application execution

In the figure, the gray part represents the additional circuitry supporting fault tolerance, that will be described in Section VI. The MPH embeds a set of memory-mapped registers that are programmed by the processor, to control send and receive operations, setting the previously described parameters.

It also includes an address generator in charge of generating the addresses when the private memories must be accessed from the port reserved for message passing.

When the processor wants to call a *send()*, the microcode that implements the primitive stores the required values into the send-related memory-mapped registers. As soon as the registers are programmed, the *address generator* starts to load *SendDim* words from the memory, starting from address *SendAddr*, and propagates them to the NI. The destination address requested for the network transaction is obtained by the *address generator* according to the content of *SendID*, translating the destination process ID into the network address of the destination processor private memory.

At the other end of the communication, the processor needs to execute a *receive()* to complete the transaction. It may happen that the *receive()* has not been called at the moment the packets composing the message actually arrive to the destination network node. In this case the message data is stored in the memory, inside a (configurable) memory buffer reserved for such a purpose. The identification fields related to the incoming message (sender, tag, buffer address) are stored inside an event file, in order to enable the *receive()* primitive to retrieve the message from the memory when it will eventually be executed. The *receive()* microcode, as a first step, stores the parameters inside three memory-mapped registers. Once such registers are programmed, the processor must keep accessing the DMA, scanning the event file locations, to check if the message under reception is already inside the buffer. In the case of a match, the processor copies the message data from the buffer to the *ReceiveAddress*. If the message is not found in the event file, the processor keeps polling the DMA handler, where a dedicated circuitry is in charge of comparing the incoming messages with the contents of the three registers. In case of matching, the message data is stored in memory, directly at the location identified by *ReceiveAddress*. In order

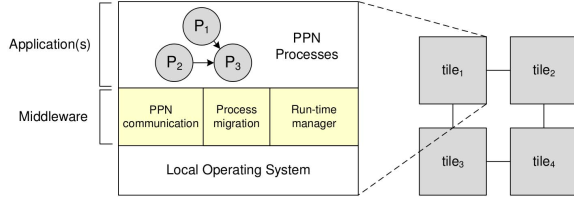

Fig. 2. Proposed software stack in MADNESS

to allow partial buffer de-fragmentation, the buffer is treated as a list.

C. Interrupt generation support

A tag decoder has been instantiated inside the Network Adapter. It is in charge of detecting a set of pre-determined tag configurations, that are reserved for the purpose of remote interrupt generation. In case of matching, the tag decoder triggers an interrupt signal that is connected to the processor interrupt controller. This feature can be used to allow a processor in the system to generate an asynchronous event on another processor, such as, for example, the initiation of the migration process.

V. Software infrastructure

Each tile of the system described in Section IV is endowed with the software stack depicted in Fig. 2. The *application level* resides at the top of the software stack. In MADNESS, applications are specified using the Polyhedral Process Networks (PPNs) model of computation. PPNs represent a special class of Kahn Process Networks, and are composed of concurrent processes that communicate using bounded FIFO channels. The PPN semantics forces a process to *block on read*, when trying to read a data token from an empty FIFO, and *block on write*, when trying to write data to a full FIFO. The simple operational semantics of PPNs allows for an easy adoption of system adaptivity and fault tolerance policies. For instance, the process state that has to be transferred upon process migration does not have to be specified by hand by the designer and can be smaller compared to other solutions.

At the bottom of the software stack, the *local operating system* provides basic functionalities such as process management (process creation/deletion, setting process priorities) and multitasking capabilities.

The *middleware level* of the software stack, highlighted in the left part of Fig. 2, comprises the three main components described below, in Section V-A, V-B and V-C.

A. PPN communication API

Based on the programming model described in Section IV-A, the PPN communication API provides a set of primitives which allow the execution of applications modeled as PPNs on NoC-based MPSoC platforms. In particular, this API must enforce the semantics of the PPN model of computation over NoC implementations with no direct remote memory access, as the one considered in MADNESS.

Several methods to implement the PPN communication over NoC-based MPSoCs are described in [10], namely *Virtual Connector, Virtual Connector with Variable Rate, and Request-driven*. However, in MADNESS we adopt the *Request-driven* communication approach as it leads to an easier implementation of the migration mechanism due to the reduced number of synchronization points between processes.

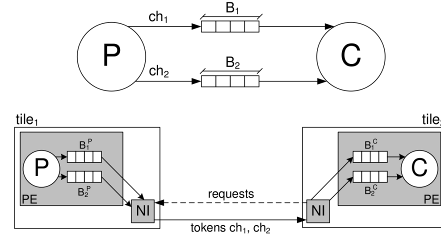

Fig. 3. Producer-consumer inter-tile communication implementation

An example of a PPN producer-consumer processes communicating over a NoC is shown in Fig. 3. In the *Request-driven* approach, each FIFO buffer of the original PPN graph is split into two buffers, one on the producer tile and one on the consumer tile. For instance, B_1 in the top part of Fig. 3 is split in B_1^P on $tile_1$ and B_1^C on $tile_2$. The size of these buffers is set such that, for all channels B_i in the original graph, $B_i^P = B_i^C = B_i$. Moreover, the transfer of tokens from the producer tile to the consumer tile is *initiated by the consumer*. This means that every time the consumer is blocked on a read at a given FIFO channel, it sends a *request* to the producer to send new tokens for that channel. The producer, after receiving this request, sends *as many tokens* as it has in its software FIFO implementing that channel.

Interrupt-based request messages: In [10] we implemented the *Request-driven* approach using a *passive* middleware. This means that the synchronization protocol was implemented by polling the Network Adapter buffer on each tile to fetch incoming requests and then react consequently. Compared to [10], we have extended the architectural support for the *Request-driven* approach. With the mentioned extension, request messages generate an interrupt on the producer tile. In this case the interrupt handler can serve the incoming request immediately.

This interrupt-based implementation of the handshake has several advantages. For instance, it relieves the processor from the burden of periodically performing non-blocking receives to check for requests incoming from the successor processes. Moreover, the asynchronous trigger can improve the predictability of the communication scheduling. Request messages can be served at any time, as soon as they arrive at the producer tile, improving the communication and computation overlapping. However, in the passive middleware, sending data only at fixed points during the execution allows easier control of the state of the handshake in case of task migration. A preliminary assessment of the effectiveness of the interrupt-based request mechanism is presented in Section VII-A.

B. Process migration mechanism

In the MADNESS project we have developed and evaluated a predictable and reliable process migration mechanism which is briefly described in the following. A simple example of a

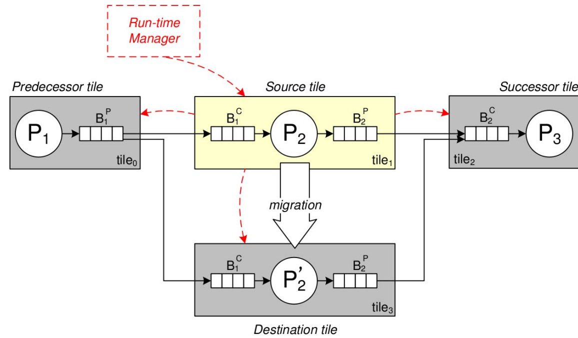

Fig. 4. Migration scenario

process migration scenario is depicted in Fig. 4. The figure shows the tiles directly involved in the process migration procedure, which are:

- the *source tile*, namely the tile which runs the process before the migration;
- the *destination tile*, which is the tile that will execute the process after the migration;
- the *predecessor tile(s)*, which runs the predecessor process(es);
- the *successor tile(s)*, which executes the successor process(es).

The structure of the PPN process has been modified to allow migration at any point within the process main body. For further details refer to [10].

The migration mechanism requires actions from all the tiles depicted in Fig. 4. The migration decision, namely which process has to be migrated and where, is taken by the resource manager using the policies described in Section VI-C. Then, the resource manager sends a specific control message to the source tile. The source tile broadcasts this control message to the destination, predecessor and successor tiles to complete the migration procedure.

For each of the tiles involved in the migration procedure, the detailed list of required actions are explained below.

1) Actions on the source tile: On the source tile, the process has to be stopped, and its state saved and forwarded to the destination tile. For a given PPN process, the state is composed only of its input and output FIFO buffers and its iterator set. The iterator set is a set of variables which defines the current iteration of the PPN process. The source tile takes also care of propagating the migration decision to the other tiles involved in the migration procedure. This propagation is depicted by the dashed arrows in Fig. 4.

2) Actions on the destination tile: The destination tile receives a specific message for process activation. The migration procedure is handled by creating the required software FIFOs and by activating the replica of the migrated process using the corresponding OS call. Before the process replica is started, the state of the migrated process is resumed. This implies that the input and output FIFOs of the migrated process are copied, and the execution starts from the beginning of the iteration that has been interrupted on the source tile.

3) Actions on predecessor and successor tile(s): On these tiles, the only required step is the update of the middleware tables where the current mapping of the processes in the system is stored. This way, new tokens or new requests meant for the migrated PPN process will be sent to the destination tile.

C. Run-time manager

The run-time manager is the entity which makes decisions concerning the adaptation of the system to changing resource availability and/or changing user requirements. In the MADNESS project the developed run-time manager is focused on fault tolerance and uses the techniques described in Section VI-C. In this context, the main responsabilty of the run-time manager is to decide to which resources to migrate the processes running on a tile which experiences a permanent fault. However, the tasks of the run-time manager may be different, according to the desired goals.

VI. Fault tolerance support

The MADNESS project focuses on the development of fault tolerance solutions which are not dependent on a technology-related low-level fault model, but rather on technology-abstracting functional-level error models. The implemented fault tolerance approaches focus on the detection of run-time faults and on the use of reconfiguration strategies at different levels. In the MADNESS framework, three main types of components are considered, i.e., *processing cores*, *storage elements*, and *NoC modules*. In this paper, the solutions proposed for the case of processing cores are described.

A. Fault detection

For the detection of faults in the processing cores, one of the two approaches are used depending on the criticality of the application.

1) Self-testing module: If the application is not critical and a limited amount of error propagation is acceptable, a self-testing routine is executed periodically by the processing element to detect its permanent faults [11]. The self-testing module (shown in grey in Fig. 1) calculates a signature of the results of the execution of the software routine, and compares it with a pre-calculated and pre-loaded correct signature. In case of a mismatch, a *fault* detection signal is raised. The self-testing routine should have a high fault coverage, a small code size and a fast execution time.

2) PPN-level self-checking patterns: For critical applications, concurrent self-checking techniques are employed at the process network level [12]. In the case of the *N-modular redundancy* (NMR) pattern, N instances of the same task are created and guarded within a *fork* and a *voter* task. The fork task simply forwards same copies of the token to each redundant instance of the task, whereas the voter task determines the most recurring result produced by the redundant task instances. For $N \geq 3$, the voter is able to detect the faulty node and mask the error. In order to yield higher reliability, the redundant instances should be mapped onto different processing elements. The task graph can be transformed with patterns in various ways leading to different levels of reliability.

B. Task migration hardware module

Task migration can be used as a reconfiguration mechanism to survive in presence of faulty processing cores. However one fundamental restriction in such a scenario is that the faulty processor cannot aid in carrying out the migration procedure. As a remedy to this problem, a task migration hardware (TMH) module is proposed which is responsible for extracting the critical data from the faulty tile.

As shown in Fig. 1, the TMH resides alongside the network adapter of each tile. It receives a fault detection signal from the self-testing module. Upon the detection of the fault, the TMH initiates the migration procedure that consists of the following steps:

- the TMH isolates the faulty processing core,
- the TMH notifies the run-time manager (RM) that resides on a fault-free core,
- the RM calculates the new mapping of the tasks,
- the RM asks the TMH for tasks' state and FIFO tokens,
- the TMH sends tasks' state and FIFO tokens to the RM,
- the RM carries out the software-based task migration as described in Section V-B.

The main figure of merit adopted when designing this module has been circuit complexity, so as to guarantee that failure rate will be much lower than the processing core. Moreover, the TMH and the software-based task migration procedure are loosely coupled such that the modifications to the software-based task migration procedure in the later stages would not affect the functionality of the TMH, thus incurring minimal changes to the TMH, if any.

C. Online task remapping strategies

A fundamental step in the fault tolerance support is deciding the new cores where the tasks formerly executed by the faulty cores shall continue their execution. In order to provide a graceful degradation, the remaining fault-free cores of the platform should be used as optimally as possible. The remapping problem can be solved by an exhaustive analysis done at design-time that evaluates all possible fault scenarios of the system and embeds in the memory the optimal remapping results to be used when faults are encountered. An alternative approach is using online task remapping heuristics, whereby the decision is taken by a remapping heuristic executed at run-time. Such an approach requires less memory, does not require a heavy design time analysis and can work even if the application running on the platform is not known *a priori*. However the degradation estimations may not be as accurate due to the usage of an analytical model rather than more detailed simulation models. In the MADNESS project, we have been investigating both approaches. However, in this paper, we adopt the online heuristics approach, in particular, the NMS-A heuristic proposed in [13].

The heuristic can be summarized as follows: let L_j be the set of tasks assigned to core n_j. L_f is the set of tasks to be migrated from the faulty node n_f. T_j^N is the sum of the execution times of tasks assigned to node n_j. $T_{cap\ ij}^{TN}$ is the execution time of task t_i if assigned to node n_j. The task $t_i \in L_f$ is remapped on the core that minimizes its finishing time. Inputs to the NMS-A algorithm are the initial mapping L, faulty node set n_f, and T^N before the fault occurrence. The output is the new mapping L. The heuristic is implemented on the MADNESS platform as a part of the run-time manager shown in Fig. 2, and it is called upon the reception of the fault detection message from the TMH.

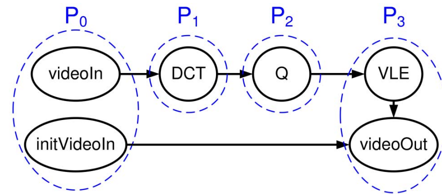

Fig. 5. PPN specification of the M-JPEG encoder.

TABLE I
EXECUTION TIMES OF M-JPEG PROCESSES

Process	Execution time (c.c.)
P_0	1923
P_1	123626
P_2	69254 (avg)
P_3	47989 (avg)

VII. EXPERIMENTAL RESULTS

To evaluate the integration of the implemented system adaptivity and fault tolerance techniques, we present an experiment that refers to a benchmark case-study application. The application is mapped on a 2x2 mesh of general-purpose processors implemented on an FPGA board. Firstly, we verify that the PPN communication API enables inter-tile communication and we compare the *passive* and *active* implementation of the middleware. Then, we present a remapping process, exploiting the migration mechanism according to the on-line remapping strategies. Finally, we test the accuracy of such strategies to verify the optimality of the chosen migration pattern. We used the M-JPEG encoder application as a benchmark application. Its PPN specification is shown in Fig. 5. The size of tokens ranges between 16 and 1024 bytes, and all of the channels are written 128 times, except the output of *initVideoIn* which is written only once. Fig. 5 also shows how some processes have been merged to map the application on the NoC platform, e.g. *VLE* and *videoOut* processes have been merged into process P_3. The numbers of clock cycles required for the execution of each process of the M-JPEG application are summarized in Table I. These numbers show that this application has a high computation/communication ratio.

A. Flow control functionality assessment

Mapping the application on the hardware platform allowed us to test the functionality of the PPN communication APIs. As mentioned, the M-JPEG application is computation-intensive, so communication latencies due to the flow control do not have a deep impact on the overall performances [10]. We tested the *Request-driven* flow control by comparing the previously proposed approach with interrupt-based implementation. The

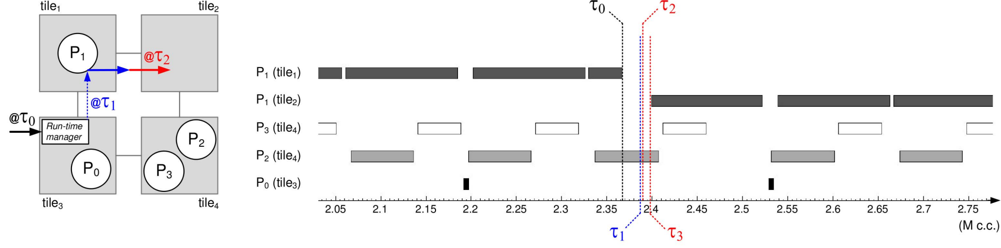

Fig. 6. M-JPEG process scheduling when migrating P_1 using the proposed remapping heuristic and migration mechanism.

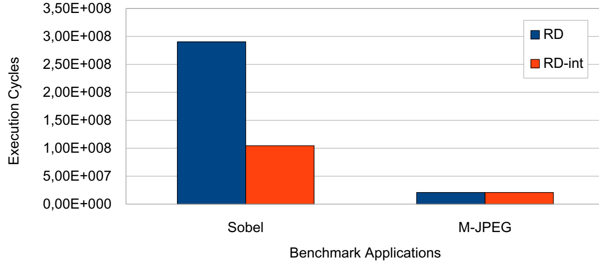

Fig. 7. Impact of the interrupt-based request messages on the *Request-driven* flow control on two benchmark applications.

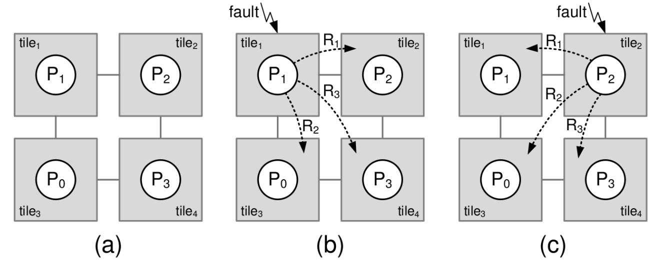

Fig. 8. Initial mapping and the two single fault scenarios showing all possible remappings.

two approaches do not lead to significant differences in the M-JPEG case study, as shown in Fig. 7. Thus, in order to compare them over a more communication-intensive benchmark, we repeated the experiment executing a Sobel filtering kernel on the platform. In this case, the execution time was significantly reduced (ca. 64%), as expected.

B. Remapping heuristic and process migration overhead

We evaluate the proposed process migration mechanism and remapping heuristic overhead using the setup shown in the left part of Fig. 6. The processes $P_0 - P_3$ in the figure refer to the specification represented in Fig. 5. Initially, P_0 is mapped on $tile_3$, P_1 on $tile_1$, P_2 and P_3 on $tile_4$. This process mapping results in a total execution time of the M-JPEG application of $T_{exe}(noMig) = 17,332,807$ clock cycles (c.c.) if no migration is performed.

However, in this experiment at time τ_0 we trigger an interrupt on $tile_3$, which activates the *run-time manager*. This interrupt emulates a message sent from the TMH of $tile_1$, indicating that the processing element is faulty. Thus, the processes running on it have to be migrated somewhere else. The migration procedure is then started. It can be divided in the timing intervals shown in the right part of Fig. 6 and described below.

- $[\tau_0, \tau_1]$**:** this is the time required by the *run-time manager* to make the remapping decision
- $[\tau_1, \tau_2]$**:** in this time interval the *source tile* ($tile_1$) sends all the process state to the *destination tile*
- $[\tau_2, \tau_3]$**:** between these two instants the *destination tile* ($tile_2$) copies the process state to its local memory and starts the execution of the migrated process

In total, the migration procedure takes $(\tau_3 - \tau_0) = 28,934$ clock cycles. Note that the execution of the migrated process has to be restarted from the beginning of the interrupted iteration. Thus, all the time spent since the beginning of the interrupted iteration on $tile_1$ has to be added to the total overhead. The worst case overhead due to the re-execution of the interrupted iteration is as large as the execution time of a whole iteration of the interrupted process, in this case P_1. The worst-case total overhead in the scenario depicted in Fig. 6 then grows up to $(\tau_3 - \tau_0) + T_{exe}(P_1) = 152,560$ clock cycles. Compared to the total execution time of the application without migration ($T_{exe}(noMig)$), this represents only 0.88% of the time.

Note that the TMH modules are not integrated yet in the MADNESS reference platform. This is why we have to emulate their behavior in software. However, we argue that the migration overhead calculated above for the software implementation will still remain small, and likely smaller, in case of a hardware implementation of TMH modules.

C. Evaluation of the remapping strategy

In this section, the quality of the heuristic is evaluated using the M-JPEG case study by comparing the remapping obtained by the NMS-A heuristic with actual measurements. Given a 2x2 NoC-based platform with processing elements n_1, n_2, n_3, n_4 and an initial mapping of M-JPEG tasks I : $P_0 \to n_3$, $P_1 \to n_1$, $P_2 \to n_2$, $P_3 \to n_4$ as shown in Fig. 8(a), we consider two single fault scenarios for n_1 and n_2. As shown in Fig. 8(b), for the case of n_1 faulty, all possible remappings are R_1 ($P_1 \to n_2$), R_2 ($P_1 \to n_3$) and R_3 ($P_1 \to n_4$). Similarly, Fig. 8(c) shows the case of n_2 faulty for which all possible remappings are R_1 ($P_2 \to n_1$), R_2 ($P_2 \to n_3$) and R_3 ($P_2 \to n_4$). The total execution times of the M-JPEG application for all possible remappings, T_{R_i}, are measured on the platform using the RD-int flow control and also calculated

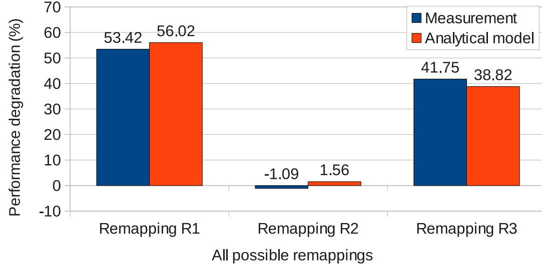

Fig. 9. Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty as shown in Fig. 8(b).

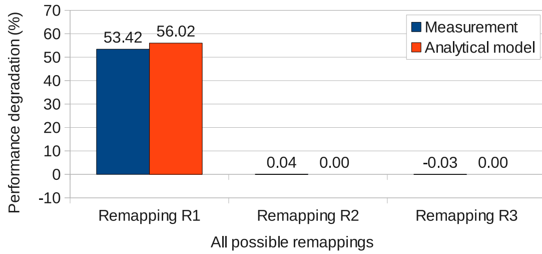

Fig. 10. Comparison of measured and calculated performance degradation of all possible remappings when n_2 is faulty as shown in Fig. 8(c).

by the analytical model.

The performance degradation with respect to the execution time of the initial mapping, T_I, is calculated according to Equation 1.

$$Performance\ degradation(R_i) = \frac{T_{R_i} - T_I}{T_I} \qquad (1)$$

Measured and calculated values are used in Equation 1 for calculating the *measured* and *analytical model* degradation results shown in Fig. 9 and 10 for faulty n_1 and faulty n_2 cases, respectively. Note that in some cases, for instance *R2* in Fig. 9, the remapping can lead to a performance speedup. In *R2*, this is because the reduction of the communication time over the NoC overcompensates the increased computational workload on n_3.

The optimal remapping is the one which yields to the smallest performance degradation. For the faulty n_1 scenario, NMS-A heuristic yields to the remapping R_2 which is the optimal decision. For the faulty n_2 scenario, it yields to the remapping R_2 which is only $.07\%$ worse than the optimal one (R_3). NMS-A makes the optimal decision according to the analytical model and the discrepancy between the analytical model and the actual measurements causes a sub-optimal decision in reality. However, as shown in Fig. 9 and 10, the analytical model estimates the degradation within 3% of the measured values. The inaccuracy of the analytical model is due to the blockings in the communication and the unaccounted context switching times when several tasks are running on a processor.

VIII. Conclusions

This paper presents the methods developed within the MADNESS project to allow system adaptivity and fault tolerance on NoC-based MPSoCs. The proposed approach involves different layers of the system design. At the application level, the PPN MoC has been selected, due to its simple operational semantics and the facilitation of system adaptivity mechanisms. At the middleware level, we have developed a communication approach to implement inter-tile PPN communication and a predictable process migration mechanism. At the hardware level, the platform has been extended in order to support the PPN MoC and to enable a predictable and efficient process migration mechanism. The process migration mechanism, in turn, can be exploited by the run-time manager to cope with permanent faults by migrating the processes running on the faulty processing element. A fast heuristic is used to determine the new mapping of processes to tiles. We show in a real-life case study that this heuristic is able to find near-optimal remappings. Moreover, the experimental results prove that the overhead due to the execution of the remapping heuristic, together with the actual process migration, is almost negligible compared to the execution time of the whole application. This means that the proposed approach allows the system to react to faults without a substantial impact on the user experience.

References

[1] E. Cannella, L. D. Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer, and A. D. Pimentel, "Towards an esl design framework for adaptive and fault-tolerant mpsocs: Madness or not?" in *Proc. of the 9th IEEE/ACM Sym. on Embedded Systems for Real-Time Multimedia (ESTIMedia'11)*, October 2011.

[2] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. Emb. Sys.*, vol. 2007.

[3] V. Nollet, D. Verkest, and H. Corporaal, "A Safari Through the MPSoC Run-Time Management Jungle," *Signal Processing Systems*, vol. 60, no. 2, pp. 251–268, 2010.

[4] G. M. Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, and M. Robert, "An Adaptive Message Passing MPSoC Framework," *Int. J. of Reconfigurable Computing*, vol. 2009, p. 20, 2009.

[5] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proc. of the conf. on Design, automation and test in Europe*, ser. DATE'06, 2006, pp. 15–20.

[6] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications," *EURASIP J. Emb. Sys.*, vol. 2008, 2008.

[7] C. Lee, H. Kim, H.-w. Park, S. Kim, H. Oh, and S. Ha, "A task remapping technique for reliable multi-core embedded systems," in *Proc. of the 8th Int. Conf. on Hardware/software codesign and system synthesis*, 2010, pp. 307–316.

[8] C.-L. Chou and R. Marculescu, "Farm: Fault-aware resource management in noc-based multiprocessor platforms," in *Design, Automation Test in Europe Conf. Exh. (DATE), 2011*, march 2011, pp. 1 –6.

[9] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "Xpipes: a Latency Insensitive Parameterized Network-on-Chip Architecture for Multi-Processor SoCs," in *Proc. of the 21st Int. Conf. on Computer Design*, ser. ICCD'03, Washington, DC, USA, 2003, pp. 536–.

[10] E. Cannella, O. Derin, P. Meloni, G. Tuveri, and T. Stefanov, "Adaptivity Support for MPSoCs based on Process Migration in Polyhedral Process Networks," *VLSI Design*, vol. 2012, no. Article ID 987209, p. 17 pages.

[11] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," in *43rd Design Automation Conf.,*, 2006, pp. 393–398.

[12] O. Derin, E. Diken, and L. Fiorin, "A middleware approach to achieving fault-tolerance of kahn process networks on networks-on-chips," *Int. J. of Reconfigurable Computing*, vol. 2011, no. Article ID 295385, p. 15pg.

[13] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Proc. of the 5th ACM/IEEE Int. Sym. on Networks-on-Chip*, 2011, pp. 129–136.