Microprocessors and Microsystems 37 (2013) 515–529

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project

Onur Derin [a], Emanuele Cannella [b], Giuseppe Tuveri [c], Paolo Meloni [c,*], Todor Stefanov [b], Leandro Fiorin [a], Luigi Raffo [c], Mariagiovanna Sami [d]

[a] ALaRI, Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland
[b] LIACS, Leiden University, 2333 CA Leiden, The Netherlands
[c] DIEE, University of Cagliari, 09123 Cagliari, Italy
[d] Politecnico di Milano, Dipartimento di Elettronica e Informazione, Milano, Italy

ARTICLE INFO

Article history:
Available online 2 August 2013

Keywords:
Embedded systems
MPSoCs
Network-on-Chip
Task migration
Fault-tolerance

ABSTRACT

Modern embedded systems increasingly require adaptive run-time management of available resources. One method for supporting adaptivity is to implement run-time application mapping. The system may adapt the mapping of the applications in order to accommodate the current workload conditions, to balance the computing 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 focused 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 regarding the system adaptivity and fault-tolerant processing. We report the results of the integration between platform level and middleware level support for adaptivity and fault-tolerance. Two case studies demonstrate the survival ability of the system via a low-overhead process migration mechanism and by taking near optimal remapping decisions at run-time.

1. Introduction

Modern technology nodes provide huge integration capabilities. This allows the multi-processor paradigm to be frequently and effectively applied in the embedded systems domain. Embedded system designers can compose computing platforms featuring an ever-increasing number of processing elements and functional blocks, placed on a single silicon die. Within the MADNESS project, an integrated framework for the application-driven design of MPSoCs was studied and implemented, aimed at supporting the designer during such a complex process. The framework is composed of several tools and IPs, interacting to achieve the identification and the implementation of the optimal system configuration within the context of streaming applications. 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 focus on the description of the mechanisms and methodologies that were defined within MADNESS for achieving run-time migration of processes among tiles, and for exploiting the proposed reconfiguration strategies in the case of faults involving processing elements. Their implementation required the integration of newly developed IPs and techniques at both platform level and middleware level, that will be introduced more in detail in the next section and described in those following. A more detailed description of the MADNESS framework as a whole can be found in [1].

2. The MADNESS project approach to fault-tolerance and adaptivity

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

* Corresponding author.
E-mail addresses: derino@alari.ch (O. Derin), cannella@liacs.nl (E. Cannella), tuveri@diee.unica.it (G. Tuveri), paolo.meloni@diee.unica.it (P. Meloni), stefanov@liacs.nl (T. Stefanov), fiorin@alari.ch (L. Fiorin), luigi@diee.unica.it (L. Raffo), sami@elet.polimi.it (M. Sami).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.07.007

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 the case of malfunctioning.

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 an 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, referring to a message-passing model of computation. Extensions will be described more in detail in Section 4.

At the software level, a specific layered infrastructure has been devised, that enables the execution of applications described by using the Polyhedral Process Network (PPN) model of computation [2]. PPNs are composed of concurrent and autonomous processes that communicate between each other by using bounded FIFO channels. PPNs were chosen for several reasons. First, 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. Second, in PPNs the control is completely distributed, as well as the memories. This represents a good match with the emerging MPSoC architectures, in which processing elements and memories are usually distributed. Third, our approach exploits the `pn` compiler [3] to automatically convert static affine nested-loop programs (SANLPs) to parallel PPN specifications and to determine the buffer sizes that guarantee deadlock-free execution.

A middleware layer allows the implementation of the PPN semantics onto the lower-level APIs provided by the architecture. In addition, it is actually in charge of managing the mapping of the PPN tasks, the communication between them and the migration process. The software/middleware infrastructure will be described in Section 5.

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 the 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. The reasons behind this choice derive from the constraints presented by MPSoCs. While in other architectural domains fault-tolerance support could be overcome by massive redundancy, this solution is surely not applicable in our case, due to unaffordable hardware overhead. On the other hand, considering a multi-core chip in which multiple instances are present for each type of processing element, the proposed technique requires moderate structural redundancy. 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, fault recovery must enabled by a fault-aware run-time environment. Thirdly, given that a faulty processor cannot participate in the remapping process, dedicated hardware is needed to ensure the migration functionality to survive in the 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 6.

3. Related work

A survey regarding the state-of-the-art in run-time management is provided in [4], 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 the case of faults.

In [5], 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 by using a completely distributed task migration policy over a purely distributed-memory multiprocessor. Similarly to our approach, their platform is programmed by using a process network model of computation. However, our work is fundamentally different because it enables the possibility to perform migration 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 [5] the process migration is enabled only at fixed points during the execution of processes.

Dynamic task remapping is also performed in [6,7] 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 [6,7] 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 [2] with the goal of throughput minimization on multi-core 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 [9], a system-level fault-tolerance technique for application mapping aiming 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 [9], 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, by exploiting in this way the unused computing resources available in the other processing elements of the system.

4. 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 $\times$pipes-lite library of synthesizable components [10]. 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 (NIs), that are in

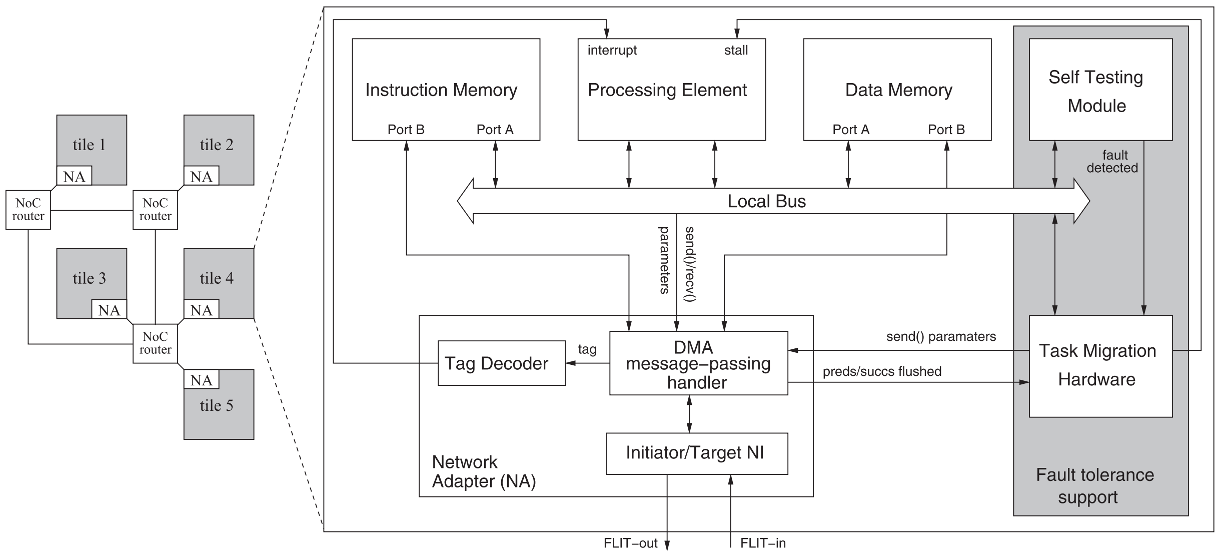

Fig. 1. A general overview of an example template instance.

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 4.2. The grey part in Fig. 1 highlights the modules devoted to fault-tolerance support. The self-testing module (STM) is a hardware module in charge of cooperating with the processing element by supporting the execution of software testing routines for detecting permanent faults in the processor. In the case of the detection of a fault, the task migration hardware (TMH) is responsible for extracting the critical data from the tile and for supporting the migration of the tasks running on the faulty processor to a fault-free one. The details of the fault-tolerance support are described in Section 6.

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

4.1. 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 4.2. 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 the 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.

4.2. Message passing support

The *Network Adapter* architecture is depicted in Fig. 1 (left side). To achieve higher performances, both the instruction and data private memories of the processor have two access ports (this feature is natively available in FPGA devices), 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 the 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.

The local bus is also in charge of managing the bus arbitration, when using single-port memories. 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 code 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 ()* code, 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 the case of a match, the message data is stored in memory, directly at the location identified by *ReceiveAddress*. In order to allow partial buffer de-fragmentation, the buffer is treated as a list.

4.3. 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 the case of match, 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 the initiation of the migration process.

5. Software/middleware infrastructure

Each tile of the system described in Section 4 is endowed with the software/middleware stack depicted in Fig. 2. The *application level* resides at the top of the software stack. In MADNESS, applications are specified by 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 by 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.

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 in the following Section 5.1, 5.2 and 5.3.

5.1. PPN communication API

Based on the programming model described in Section 4.1, 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 [11], 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, thanks to the reduced number of synchronization points between processes of this approach.

An example of a PPN producer–consumer processes communicating over an 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 at the producer tile and one at 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.

5.1.1. Interrupt-based request messages

In [11] we implemented the *Request-driven* approach by 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 [11], we have extended the architectural support for the *Request-driven* approach. With the mentioned extension, request messages generate an interrupt at 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 the case of task migration. A preliminary assessment of the effectiveness of the interrupt-based request mechanism is presented in Section 7.2.

Note that although the *interrupt-based* approach improves the predictability of the communication compared to the *passive*

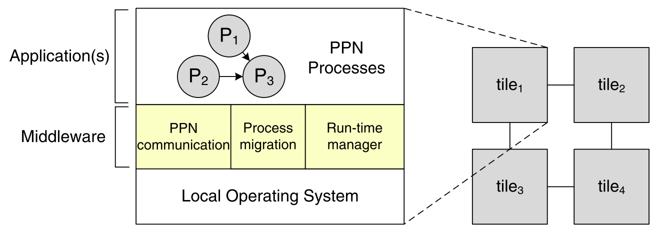

Fig. 2. Proposed software stack in MADNESS.

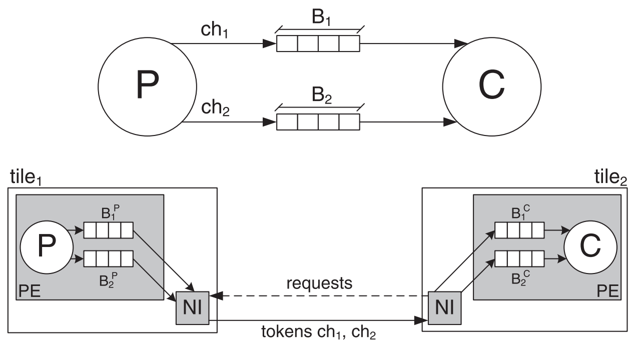

Fig. 3. Producer–consumer inter-tile communication implementation.

request-driven approach, they both introduce a latency in the communication between two processes running on different tiles. A direct point-to-point connection between the two considered tiles would result in faster inter-tile communications. However, this communication efficiency would come at the price of a much reduced system adaptivity.

5.2. 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. The process migration mechanism is based on mainly two starting assumptions:

1. Process migration is based on *process replication*, which means that the code of the migratable tasks is copied in each tile of the system. Upon migration, only the *state* of the migrating process has to be transferred.
2. To take into account the run-time remapping of processes over the system, each tile stores in its local memory a *middleware table*, which is used to refine the generic communication primitives to mapping-dependent function calls.

Beside custom-made functions implemented in the middleware layer, the process migration mechanism relies on system calls of the underlying operating system (OS), which in our case is *Xilkernel*, provided by Xilinx. For instance, when the system is started, all the task replicas on all the tiles are created. However, only the replicas which are included in the initial mapping are activated. Process replicas which are not included in the initial mapping are created anyway at startup in order to shorten in this way the time required to activate the replica in the case of migration.

A simple example of a 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 [11].

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 by using the policies described in Section 6.2.5. 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.

5.2.1. 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. The mentioned actions can be carried out by the processing element of the considered tile, in case of migration for system adaptivity purposes. However, in case of a migration triggered by a fault in the processing element of the source tile, similar actions (with minimal modifications) have to be performed by the TMH of the faulty tile.

5.2.2. 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 by 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. Note that the copy of input/output FIFOs and the fact that the execution is started from the beginning of the interrupted iteration introduce a latency in the execution of the migrated process. However, the primary goal of our work is to guarantee the survival of the system in the case of faults, with minimal hardware overhead. We assume that the system can tolerate a certain latency in the case of a fault. Moreover,

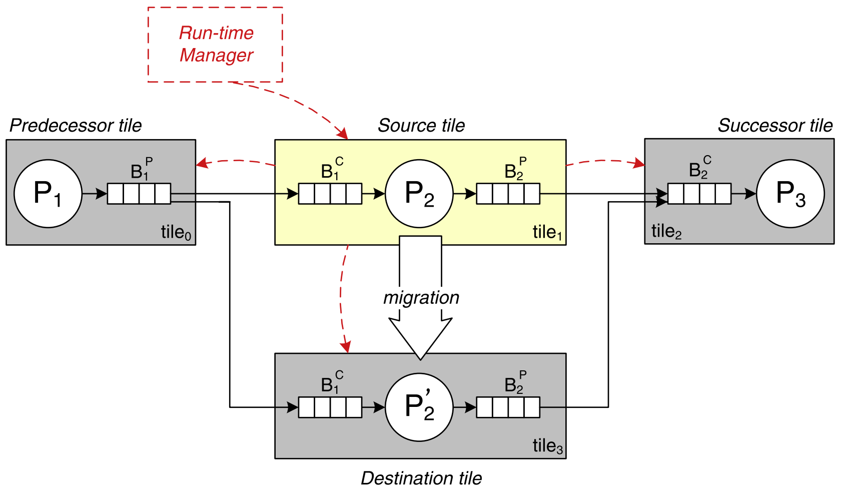

Fig. 4. Migration scenario.

the experimental results in Section 7.3 show that this latency overhead can be minimal in real-life applications.

5.2.3. 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.

5.3. 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 focuses on fault-tolerance and uses the techniques described in Section 6.2.5. In this context, the main responsibility of the run-time manager is to decide to which resources migrating 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.

6. 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, we describe the work done to enable continuity of service in the presence of permanent faults in processing elements when running PPN applications on NoC multiprocessors. As a basic assumption, online software-based self-testing is adopted for detecting permanent faults in processors. The fault model comprises stuck-at faults which may represent various types of errors at the processor level such as halting of the processor (crash), wrong computation and execution of arbitrary code in the program memory, etc. The NoC components and memories are assumed to be designed so as to grant continuity of service even in the presence of (a predetermined set of) faults such that they exhibit a much smaller IP-level failure rate. We assume that one CPU at a time can fail and that remapping has been completed before possibly a new fault appears in another CPU.

The proposed solution encompasses support for fault detection and fault recovery via online task remapping. It involves hardware and software modifications on top of the MTOS, message-passing support, and the NORMA-based NoC platform. The proposed fault recovery technique is based on rolling back the execution at the granularity of a single iteration of a PPN process. Fault detection relies on executing the self-testing routine at the end of each iteration of a process. If the test is successful, the results of the current iteration, which are to be written to the output FIFO channels of the process, are guaranteed to be correct. If the test fails, the recovery mechanism is started with the help of TMH, which is responsible for notifying the run-time manager and for transferring the state of the tasks, i.e., the iterators of the tasks and the tokens in the input and output FIFO channels. In the remainder of this section, the fault detection and recovery mechanisms are explained in detail by describing the implemented hardware and software support.

6.1. Fault detection

Depending on the criticality of the application, two different approaches can be applied for the detection of faults in the processing cores, i.e., periodic online self-testing routines, and self-checking patterns at task level [1].

6.1.1. 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 [12–14]. The *Self-testing module* (shown in grey in Fig. 1) supports the execution of testing routines in each tile of the Network-on-Chip (NoC) which includes a processing element. Hardware modules are needed whenever it is not possible to implement techniques such as distributed testing [15], which relies on the availability in the same platform of several instances of the same type of general purpose processors. The STM checks the results of the software testing routines and activates the procedures for task remapping and migration, as described in Section 6.1.1. The STM is in charge of collecting the outputs of the processor when executing the software routine, of calculating the signature of the outputs of the processor, and of checking it against the expected signature, in order to verify the correctness of the routine execution. The signature is calculated applying a cyclic redundancy check (CRC) algorithm to the expected and obtained outputs of the processor [16]. The testing routine, as well as the signature of the expected results, is stored directly in the processor local memory, and it is scheduled by the operating system – memories are by assumption protected against faults by employing standard fault tolerant techniques based on the use of Error Correcting and Detecting codes [16]. Results of the execution of the software routine are written directly into the STM.

Fig. 5 shows the architecture designed for supporting the execution of the software testing routines, which is intended to work as a wrapper around the processor for helping detecting permanent faults in it. The STM is memory-mapped on the tile's system bus and it can be directly accessed by the processor. In the *prologue* of the software testing routine, the expected signature is copied in the *slv_signature* registers. Then, the STM is activated, by writing into the *slv_start_stop* register. When active, the STM samples the outputs of the device under test (DUT) (i.e., the processor) and copy them in the *slv_data_in* registers. For each new data inserted in the *slv_data_in* registers, the CRC parallel module calculates immediately the value of the signature for the samples received up to that moment. At the end of the execution of the software routine, the STM is stopped by writing into the *slv_start_stop* register. The STM compares the value stored into the *slv_signature* registers with the final signature calculated by the CRC parallel module. If the two values does not match, the *fault_detected* signal is set to '1' for a clock cycle. The STM also supports detecting crash errors. A fault may result in the processor not executing the self-testing routine at all. A timer is introduced inside the STM to detect such errors. If the self-testing routine does not complete within the time limit, a crash error is assumed and the *fault_detected* signal is raised. Otherwise, if the self-testing routine is completed (inferred by a write into the *slv_start_stop* register), the timer is stopped until the next execution is set and started.

6.1.2. PPN-level self-checking patterns

For critical applications, concurrent self-checking techniques at process network level are employed [17]. In the case of the *N-modular redundancy* (NMR) pattern, *N* instances of the same task are created between a *fork* and a *voter* task. The fork task simply forwards same copies of the token to each redundant instance of

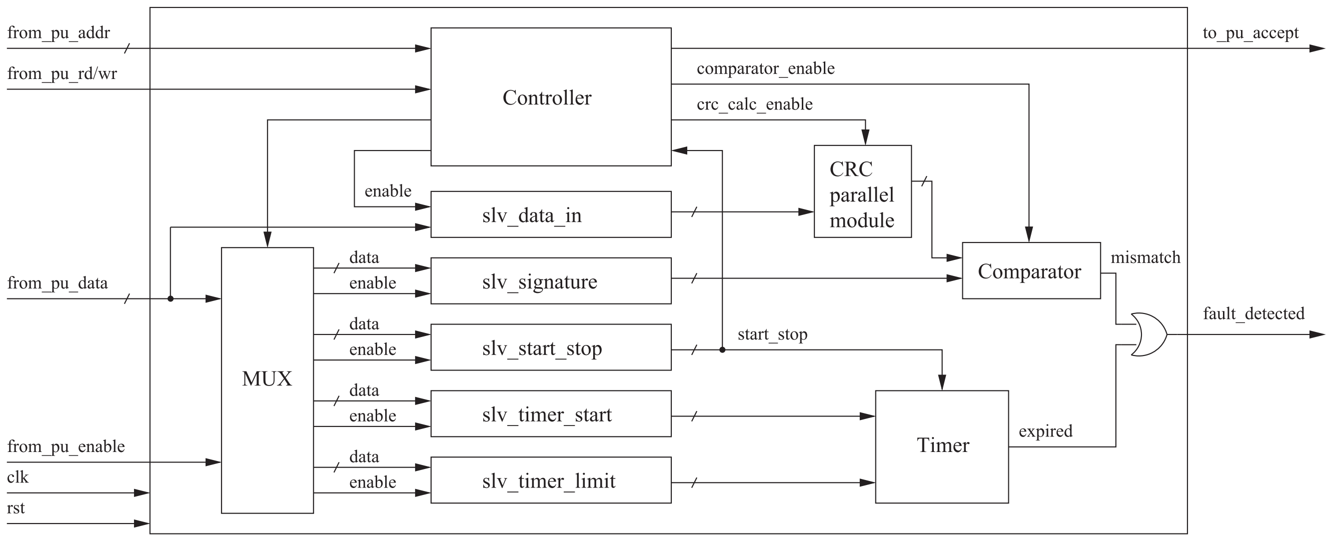

Fig. 5. Overview of the STM architecture.

the task, whereas the voter task determines the most recurring result produced by the redundant task instances. For $N \geqslant 3$, 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.

6.2. Fault recovery

The fault recovery mechanism relies on a number of changes done at the application, run-time and hardware levels.

6.2.1. Modifications to the PPN processes

A part of the fault tolerance support involves the modification of process bodies. Algorithm 2 shows how the basic process body shown in Algorithm 1 is modified to support the fault recovery mechanism.

Algorithm 1. A basic PPN process

```
1: for (i = 0; i < M; i++) do
2:    for (j = 0; j < N; j++) do
3:       read (in, CH1);
4:       out = f (in);
5:       write (out, CH2);
6:    end for
7: end for
```

All PPN processes have the same code structure (as shown in Algorithm 1). Nested loops iterate, for a given number of times, the body of the process, which is split into three main parts. First, the process reads the input data tokens from (a subset of) the input channels. This is represented by the *read*() statements in the algorithm. Second, the process function (*f*) produces the output tokens by processing the input tokens. Finally, the output tokens are written to (a subset of) the output channels represented by the *write*() statements.

Algorithm 2. The PPN process template for the proposed fault tolerance mechanism

```
1: if (migration) resumeState;
2: for (i = i_0; i < M; i++) do
3:    for (j = j_0; j < N; j++) do
4:       acqData (CH1);
5:       read (in, CH1);
6:       setTimer ();
7:       out = f (in);
8:       selfTest ();
9:       write (out, CH2);
10:      relSpace (CH1);
11:   end for
12:   reset j_0;
13: end for
```

According to Algorithm 2, when the thread starts, it checks if the *migration* flag is set (line 1). If the migration flag is false, it means that the process starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$. Otherwise, it means that a migration has been performed, so the process state is reloaded.

Since the PPN model definition requires a stateless process function (for example *f* in Algorithm 1, i.e., a function whose execution does not depend on the previous iterations), the state of a PPN process is represented only by:

- the content of its input and output FIFOs;
- its iterator set, namely the values of the nested loop iterator variables, see (i, j) in Algorithm 2, lines 2 and 3;

In functions requiring to have a state, the function state is represented in the PPN model by a stateless function with FIFO self-edges.

Due to the request-based flow control policy used for implementing the KPN semantics on the NoC platform, the pending requests on the outgoing channels from the faulty processing element also constitute in addition part of the state to be recovered. All the three state components listed above are transferred from the faulty tile to the run-time manager upon fault detection.

Lines 2 and 3 differ from the basic process structure in Algorithm 1 because the iterators inside the *for loops* do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the fault detection while running on the source tile. After the first complete execution of the inner *for loop*, starting from j_0, the value of j_0 is set to zero in line 12 such that the next execution of the inner loop starts correctly with $j = 0$.

The *read* communication primitive is different from the one used in the basic process structure. It is split into three separate operations (see lines 4, 5, 10). First, the input channel (CH1) is tested to verify the presence of an available data token, using the *acqData ()* function in line 4. Then, the token is copied from the software FIFO to the input variable which will be processed by the process function *f*. The copy operation is performed in line 5. However, differently from the normal read primitive, the memory locations occupied by the read token are not released immediately. The actual release, which consumes the data from the FIFO by increasing the read pointer, takes place only in line 10 (*relSpace (CH1)*). In this way, if a fault is detected before the release instruction, the process can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is not changed. Note that, in case of multiple input or output channels, the release operations should be grouped together and placed right after the main body of the process, in order to guarantee a consistent process state.

In order to tolerate crash errors, which result in the processor not executing any instructions, a watchdog timer is set to expire within a time limit (line 6). This time limit is greater or equal to the sum of the worst case execution time of process function (*f*) and the self-testing routine. In the case that the program counter reaches the end of the self-testing routine, the timer is reset before it times out. Otherwise, the timer signals the TMH module about the fault detection. The faults can be more subtle and may result in computational errors. Such faults are detected by running a self-testing routine as shown in line 8. In the case that the self-testing routine produces a different output than normal, it is detected by the self-testing module, which in turn signals the fault detection to the TMH.

If a crash fault occurs between the end of self-testing routine (line 8) and setting of the timer (line 6), it cannot be detected. Moreover, if a fault occurs just after executing a self-test successfully (line 8), it may result in a corrupt data to be written to the channel while executing line 9. However, we can argue that the time window (thus the probability) for such cases is very small as the majority of the time will be spent in executing the process function *f* and the self-testing routine.

6.2.2. Fault-aware remapping support

The actors involved in the fault recovery procedure are the following: (i) *processing element* of the source tile (i.e., the tile that experiences the fault); (ii) *self-testing module* in the source tile; (iii) *task migration hardware* module in the source tile; (iv) *run-time manager* which runs on one of the fault-free tiles; (v) *predecessor and successor tile(s)*: the tile(s) which has a producer or a consumer task of any of the tasks on the source tile; (vi) *new tile(s)*: the tiles that will run at least one of the tasks on the source tile after fault recovery; (vii) *other tile(s)*: the fault free tile(s) that are neither the source tile, a new tile, a predecessor or a successor tile.

After executing the self-testing routine, if a fault is detected on the source tile, the STM issues a fault detection signal to the TMH. The TMH isolates the faulty processor. The TMH reports the fault to the RM by sending a fault detection message (the selection of the RM tile is explained in Section 6.2.4). The RM calculates the new mapping of the tasks using the remapping heuristics (see Section 6.2.5). The RM informs the predecessor/successor tile(s) and the other tiles about the new mapping of the tasks to update their middleware tables. The predecessor/successor tiles send a flush message to the faulty node to make sure that there are no tokens or requests still travelling to the faulty tile. Upon the reception of flush messages from predecessor/successor tiles, the TMH responds with a flush message to make sure that there are no tokens or requests still travelling to predecessors or successors. Then the TMH sends to the RM the state of the tasks, which consists of (i) the iterators of the loops in the case of PPN tasks, (ii) the information of the FIFO channels (pending requests and number of tokens in the FIFO channels), (iii) the tokens in the input and output FIFO channels. After the RM receives the tasks' state from the TMH, the procedure goes on similar to the software-based process migration (without updating the predecessors and successors again) as described in Section 5.2. The RM sends these data to the new tile(s) according to the new mapping decision. Then the RM sends to the new tiles a task activation message along with the new mapping information allowing updating of their middleware tables and the activation of migrated tasks. This finalizes the fault recovery procedure.

6.2.3. Task migration hardware module

The task migration hardware (TMH) module is mainly 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 as input a fault detection signal from the STM. Upon the detection of the fault, the TMH carries out the following actions:

1. the TMH isolates the faulty processing core,
2. the TMH notifies the run-time manager (RM) running on the fault-free core with the nearest bigger index,
3. the TMH receives the flush messages from all predecessor and successor tiles,
4. the TMH sends the state of all tasks and channels (pending requests and FIFO tokens) to the RM,

In step 3, TMH waits for all flush messages which guarantees that the tokens (from the predecessor tiles) and the requests (from the successor tiles), which may be in transit on the NoC at the time of fault detection, are received at the faulty node before TMH sends the migration data to the RM.

The TMH module carrying out this functionality has been designed and integrated into the platform. 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.

The interface and the internal block diagram of the TMH are shown Fig. 6. The interface consists of ports allowing (i) to receive the fault detection signal from the self-testing module (*fault_detected*), (ii) to isolate the processor (*to_pu_stall*), (iii) to be read/written by the processing element from/to the register file inside the TMH (*from/to_pu_∗*), (iv) to send data via the NoC (*to_dma_∗*), (v) to receive the flush messages from predecessor and successor tiles. It consists of a control unit implementing the finite state machine, a register file, a multiplexer and shmpi_send registers. The register file contains memory-mapped registers which store (i) a pointer to the fault detection control message stored statically in the main memory, (ii) the tile ID that acts as the RM for the tile, (iii) the size of the control message, (iv) the special tag value used to send data carrying interrupt messages over the NoC, (v) the tasks mapped on the tile, (vi) the pointer to the array storing task states, (vii) the size of the task state, (viii) the special tag value used to send task states to the RM, (ix) the channels mapped on the tile, (x) the special tag value used to send channel data to the RM, (xi) a reduced middleware table containing for each channel the pointer to the software FIFO, the number of

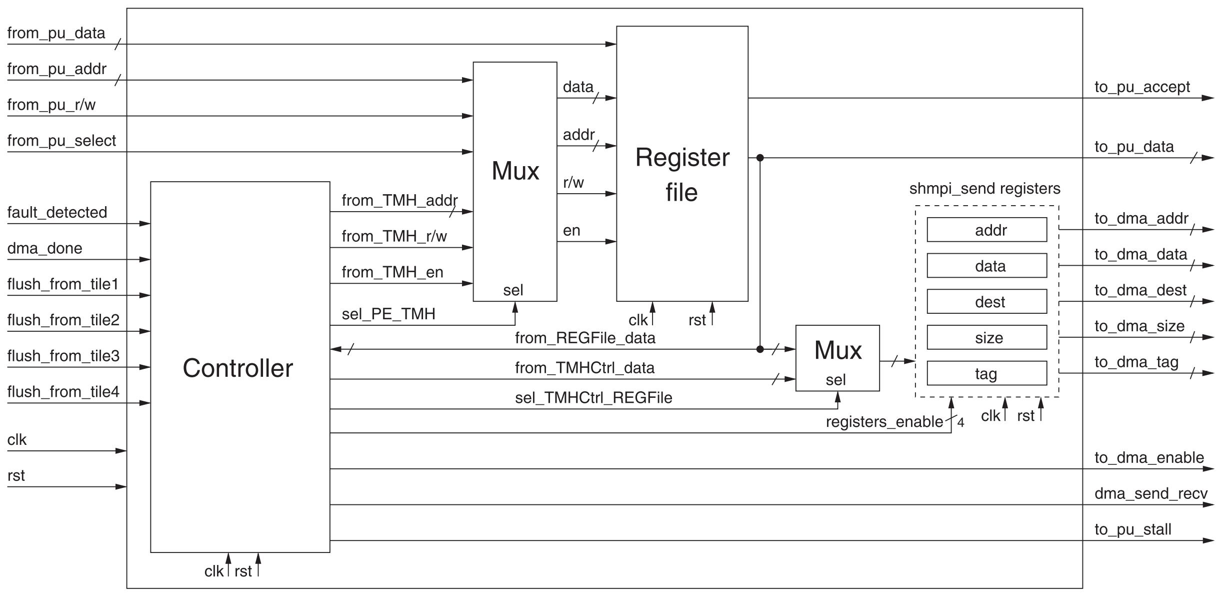

Fig. 6. Interface and internal block diagram of the task migration hardware module.

tokens in the channel, the size of the token type and a pending request flag.

When an application is launched, the PE initializes the TMH registers. During normal execution (when the PE is not faulty), whenever there is a read or a write, the number of tokens is updated in the TMH register for the corresponding channel. The read and write operations of the TMH take only cycle in order to reduce the overhead of the update operation. After the fault detection, TMH carries out a number of *shmpi_send* operations by using the programmable DMA to notify the RM, and sending states of mapped tasks and data of mapped channels.

6.2.4. Decentralization of the run-time manager

Centralized techniques represent a single point of failure and thus they should be avoided in fault tolerance mechanisms. In our fault recovery mechanism, RM is the main actor coordinating the recovery process. Therefore it is important that the RM is not centralized. As a solution to this problem, the RM is run as a dormant thread on each processing element. Any tile can assume the role of the RM when it receives an interrupting fault detection message. A simple algorithm is used to assign an RM instance to each tile. The TMH of a faulty tile sends the fault detection message to the RM instance assigned to its tile. The RM assignment algorithm is as follows: a tile chooses as its RM the tile that has the smallest index that is greater than its own index and that is fault-free. If there is no such tile, it chooses the tile that has the greatest index that is smaller than its own index and that is fault-free. This makes sure that a fault-free RM tile is assigned to each tile. Given the single fault assumption, when a fault occurs, every fault-free tile is informed about the faulty tile and update their local information about the fault status of other tiles. If the faulty tile is the currently assigned RM tile of any tile, such tiles re-run the RM assignment algorithm.

6.2.5. Online task remapping strategies

A fundamental step in the fault-tolerance support is deciding on 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 design-time analysis 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 [2]. An alternative approach is to use online task remapping heuristics [18], whereby the decision is taken by a remapping heuristic executed at run-time. Such an approach requires less memory, it does not require a heavy design time analysis, and it can work even if the application running on the platform is not known *a priori*. However, the estimation of the performance degradation may result less accurate: in fact, it is not possible to rely on detailed simulations performed at design time and analytical models of the system applications should be employed instead. In the MADNESS project, we have investigated both approaches. In this paper, we adopt the online heuristics approach, in particular, the NMS-A/B/C heuristics proposed in [18].

As an example, the NMS-A 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. According to the NMS-A heuristic, inputs to the algorithm implementing it 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 task $t_i \in L_f$ is remapped on the core that minimizes its finishing time. All of the NMS-A/B/C heuristics are implemented on the MADNESS platform as a part of the run-time manager shown in Fig. 2, and the selected one is called upon the reception of the fault detection message from the TMH.

7. Experimental results

In this section, we describe a set of experiments that we performed in order to evaluate the implemented system adaptivity and fault-tolerance techniques. The application case studies are described in Section 7.1. We map these applications onto a 2×2 mesh of general-purpose processors, as detailed in Section 4, implemented on a Virtex-6 FPGA board.

Firstly, we verify that the PPN communication API enables inter-tile communication according to the PPN semantics and we compare the *passive* and *active* implementation of the middleware. Then, we present a remapping process, exploiting the migration mechanism detailed in Section 5.2 and the fault recovery mechanism described in Section 6.2. Finally, we test the accuracy of such strategies to verify the optimality of the chosen migration decision.

7.1. Case studies

We chose as case studies two streaming applications in the multimedia domain, i.e., an M-JPEG encoder and a H.264 decoder. The two applications are described below.

7.1.1. M-JPEG encoder

The PPN specification of the M-JPEG encoder is shown in Fig. 7. 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 per frame. Fig. 7 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 M_3. The numbers of clock cycles required for the execution of each process of the M-JPEG application are summarized in Table 1. Comparing these numbers with the amount of inter-process communication one can infer that this application has a high computation/communication ratio.

7.1.2. H.264 decoder

The simplified PPN specification of this case study is shown in Fig. 8. In the final implementation, the nodes *get_data,parser*, and *cavlc* have been merged into a single process, H_0. In this case study, the size of the exchanged tokens ranges between 1 and 5000 bytes. The execution time of each process of the H.264 decoder application are shown in Table 2.

7.2. Flow control functionality assessment

Mapping the application on the hardware platform allowed us to test the functionality of the PPN communication APIs. We show the results obtained for the M-JPEG application. As mentioned earlier, the M-JPEG encoder is computation-intensive, so communication latencies due to the flow control do not have a deep impact on the overall performances [11]. We tested the *Request-driven* flow control by comparing the previously proposed approach with interrupt-based implementation. The two approaches did not lead to significant differences in the M-JPEG case study, as shown in Fig. 9. 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.

7.3. Remapping heuristic and process migration execution time overhead

We evaluate the proposed process migration mechanism and remapping heuristic overhead by using the setup shown in the left part of Fig. 10. Processes M_0–M_3 in the figure refer to the specification represented in Fig. 7. Initially, M_0 is mapped on $tile_3$, M_1 on $tile_1$, M_2 and M_3 on $tile_4$. This process mapping results in a total

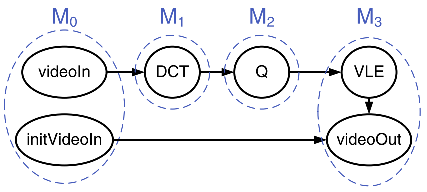

Fig. 7. PPN specification of the M-JPEG encoder.

Table 1
Execution times of M-JPEG processes.

Process	Avg. execution time (c.c.)
M_0	1923
M_1	123,626
M_2	69,254
M_3	47,989

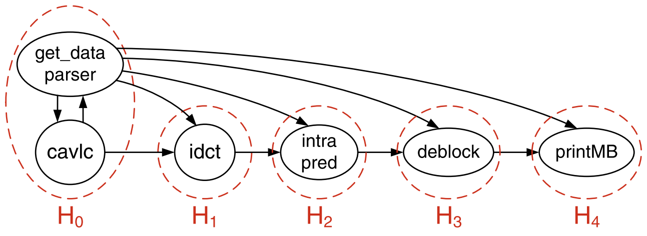

Fig. 8. Simplified PPN specification of the H.264 decoder.

Table 2
Execution times of H.264 processes.

Process	Avg. execution time (c.c.)
H_0	95,643
H_1	55,775
H_2	33,645
H_3	9724
H_4	4075

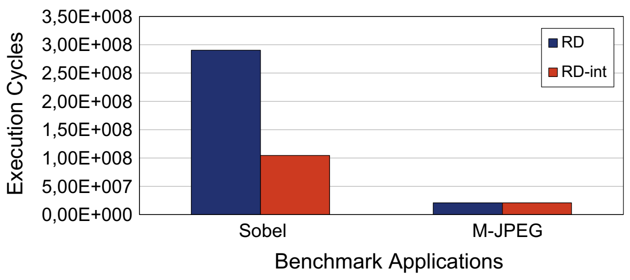

Fig. 9. Impact of the interrupt-based request messages on the *Request-driven* flow control on two benchmark applications.

execution time of the M-JPEG application of $T_{exe}(noMig)$ = 17,332,807 clock cycles (c.c.) in the case of no migration.

However, in this experiment at time τ_0 we trigger an interrupt on $tile_3$, which activates the *run-time manager*. This interrupt represents an event triggering a change of the task-to-processor mapping (e.g. a switch to a low power operating mode) and, in the presented experiment is meant to deactivate the processing core in $tile_1$. Thus, the processes running on it have to be migrated on other tiles. The migration procedure is then started. It can be divided in the timing intervals shown in the right part of Fig. 10 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

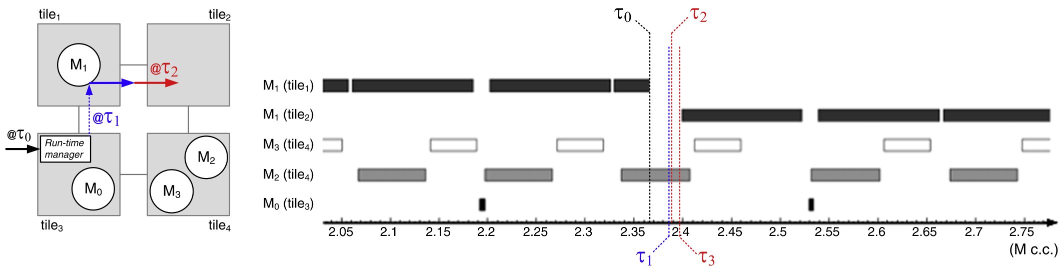

Fig. 10. M-JPEG process scheduling when migrating M_1 by using the proposed remapping heuristic and migration mechanism.

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 M_1. The worst-case total overhead in the scenario depicted in Fig. 10 then grows up to $(\tau_3 - \tau_0) + T_{exe}(M_1) = 152{,}560$ clock cycles. Compared to the total execution time of the application without migration ($T_{exe}(noMig)$), this represents only 0.88% of the time. In this experiment, we have calculated the overhead considering the remapping decision strategy described in Section 6.2.5. This allows to obtain a basis of comparison for the evaluation of the hardware-based migration mechanism that has to be used in case of fault tolerance. However, it is worth to point out that the reported migration time could be reduced using pre-calculated remapping tables, created at design time, to decide the migration pattern.

To assess the performance of the migration procedure as a fault recovery mechanism, we evaluated it with a single fault scenario (initial mapping shown in Fig. 13a) and the fault on $tile_1$ as shown in Fig. 13b. Fig. 11 shows the finishing time of each phase of the fault recovery mechanism averaged over several experiments. Time 0 corresponds to the fault detection time, i.e., activation of the TMH. The average fault recovery time is 38,115 clock cycles. 46% of this time is taken by the phase in which the state of tasks and channels from the TMH is received by the RM. In this particular scenario, the RM is also the new tile where DCT is being remapped to. Therefore, the phase of transferring the state to the new tile does not take as much time. It is worth to note that several actions of the recovery process happen in parallel, thus reducing the recovery time. For example, the data transfer from the TMH to the RM overlaps with the execution of the heuristics by the RM.

The experiment shows that the execution time of the fault recovery procedure is comparable with the duration of the software-based migration that can be used in fault-free systems. In both cases the overhead is negligible when compared with the execution time of the whole application. The increase is mostly due to some additional synchronization actions that had to be introduced in the fault recovery mechanism, to handle possible corner cases in the management of the software FIFOs. To evaluate the calculation time of the remapping decision, the two remapping scenarios given in Figs. 13 and 16 are used for M-JPEG and H.264 applications. The NMS-A/B/C heuristics from [18], which aim at minimizing the throughput degradation, are implemented on the platform. Their calculation time are displayed in Table 3. The results reveal that their execution time constitutes a relatively small portion of the fault recovery time.

7.4. Steady-state performance overhead of the fault tolerance support

There is a performance penalty that is paid in order to support fault tolerance, even in the absence of faulty processors. This is mainly due to the modifications that are done in the process bodies, in particular, the execution of the self-test at each iteration of each process. Therefore the duration of the self-testing routine influences the overhead of the technique during normal operation. Since we have not implemented real self-testing routines for the Microblaze processor, we report analytical results of this overhead with respect to various execution times of the self-testing routine ranging from 10 k to 100 k cycles. The mapping used in the calculations is the one of Fig. 13a. As shown in Fig. 12, the overhead is linear with respect to the self-test duration and changes from 7.7% to 71%. Naturally designing a self-testing routine involves a trade-off between its execution time and fault coverage ratio. Selecting 40 k cycles as a typical duration for the self-test (taken from [19] for a processor of supposedly similar complexity), we see that the overhead would be 29%. This overhead is due to additional workload inflicted upon the critical node that determines the throughput of the whole application.

7.5. Evaluation of the remapping strategy

In this section, the quality of the heuristic is evaluated by using the M-JPEG and H.264 case studies by comparing the remapping obtained by the NMS-A/B/C heuristics with actual measurements.

7.5.1. M-JPEG remappings

Given a 2×2 NoC-based platform with processing elements ($tile_1 = n_1$, $tile_2 = n_2$, $tile_3 = n_3$, $tile_4 = n_4$) and an initial mapping of M-JPEG tasks $I{:}M_0 \rightarrow n_3$, $M_1 \rightarrow n_1$, $M_2 \rightarrow n_2$, $M_3 \rightarrow n_4$ as shown in

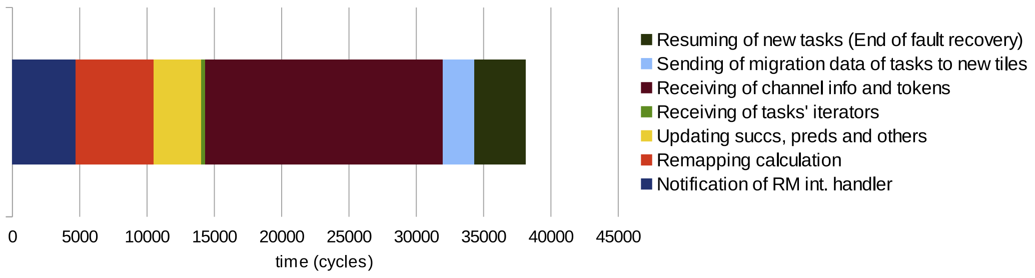

Fig. 11. Execution times of fault recovery actions.

Table 3
Calculation times of remapping heuristics.

Heuristic	Avg. execution time (c.c.)	
	M-JPEG	H.264
NMS-A	8198	8172
NMS-B	19,608	19,603
NMS-C	6403	6664

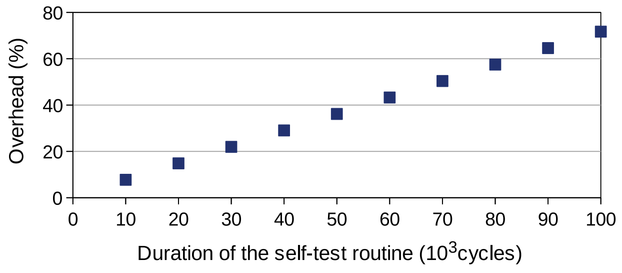

Fig. 12. Performance overhead with respect to the duration of the self-testing routine.

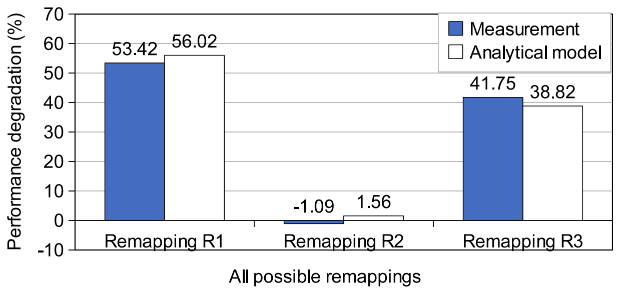

Fig. 14. Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty as shown in Fig. 13(b).

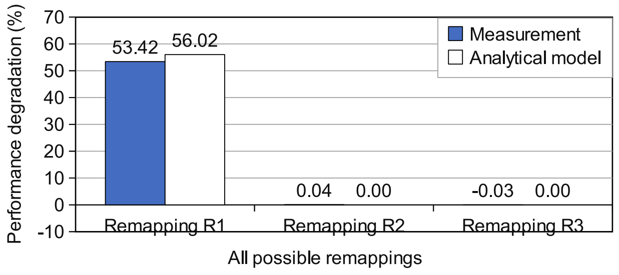

Fig. 15. Comparison of measured and calculated performance degradation of all possible remappings when n_2 is faulty as shown in Fig. 13(c).

Fig. 13a, we consider two single fault scenarios for n_1 and n_2. As shown in Fig. 13b, for the case of n_1 faulty, all possible remappings are R_1 ($M_1 \rightarrow n_2$), R_2 ($M_1 \rightarrow n_3$) and R_3 ($M_1 \rightarrow n_4$). Similarly, Fig. 13c shows the case of n_2 faulty for which all possible remappings are R_1 ($M_2 \rightarrow n_1$), R_2 ($M_2 \rightarrow n_3$) and R_3 ($M_2 \rightarrow n_4$). The total execution times of the M-JPEG application for all possible remappings, T_{R_i}, are measured on the platform by using the RD-int flow control and also calculated by the analytical model.

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

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

Measured and calculated values are used in Eq. 1 for calculating the *measured* and *analytical model* degradation results shown in Figs. 14 and 15 for faulty n_1 and faulty n_2 cases, respectively. Note that in some cases, for instance $R2$ in Fig. 14, 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, all of the NMS-A/B/C heuristics yield 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/B/C heuristics make the optimal decision according to the analytical model and the discrepancy between the analytical model and the actual measurements causes a very slightly sub-optimal decision in reality. However, as shown in Figs. 14 and 15, the analytical model estimates the degradation within 3% of the measured values. The inaccuracy of the analytical model is due to the latency introduced by the communication API (see Section 5.1) and the unaccounted context switching times when several tasks are running on a processor.

7.5.2. H.264 remappings

We use the same procedure to assess the NMS-A/B/C remapping heuristics in the H.264 case study. The initial mapping is shown in Fig. 16a. Then, we consider the case of a fault occurring either in n_1 or n_2. In each of these cases there are three possible remappings (R_1 to R_3), which are depicted in Fig. 16a and c.

In the case of a fault occurring on n_1, all of the NMS-A/B/C heuristics yield to the remapping R_3, which is the optimal one as shown in Fig. 17. In the other considered case, faulty n_2, all the heuristics suggest the remapping R_3. Also in this case, the suggested remapping represents the optimal one, as can be deduced by

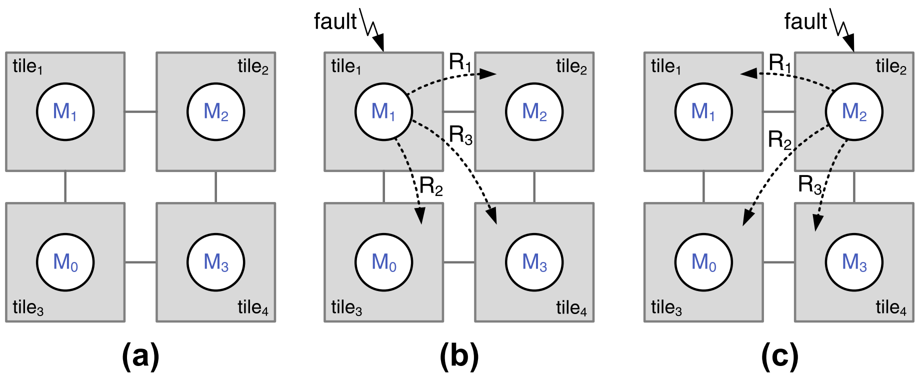

Fig. 13. Initial mapping and the two single fault scenarios showing all possible remappings.

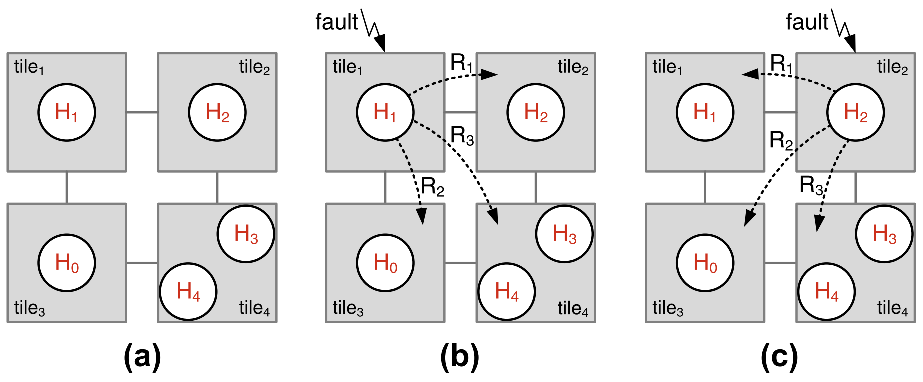

Fig. 16. Initial mapping and the two single fault scenarios showing all possible remappings.

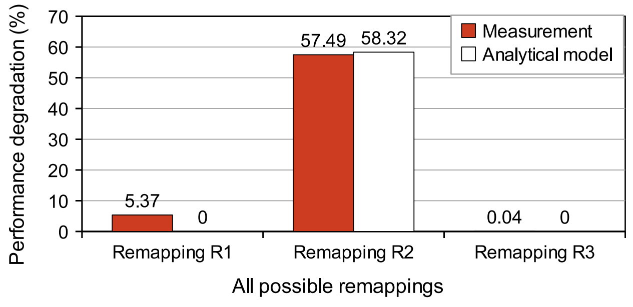

Fig. 17. Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty as shown in Fig. 16(b).

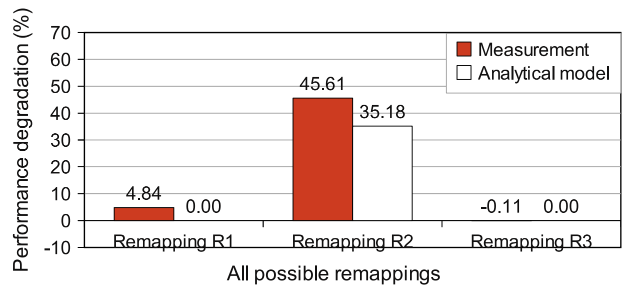

Fig. 18. Comparison of measured and calculated performance degradation of all possible remappings when n_2 is faulty as shown in Fig. 16(c).

Fig. 18. Similar to the M-JPEG experiments, the inaccuracy of the analytical model is due to the abstraction of the overheads related to context switches and communication over the platform.

7.6. Architectural support hardware overhead

Obviously, the circuitry implementing the support for adaptivity and fault-tolerance at architectural level incurs an overhead in terms of area obstruction, power consumption and critical path length. To evaluate the overhead, we consider the basic × pipes mesh as a baseline architecture. As mentioned earlier, with respect to the baseline, the Network Adapter has been enriched with the DMA message-passing handler (MPH). It provides all the message passing capabilities that are needed to implement the inter-processor communication, the triggering of the migration process and the migration process itself. Moreover, this module allows the possibility of intra-processor multitasking. Controlling the local memory, to store the incoming messages when a *receive ()* has not been performed, the MPH allows, at the producer side, scheduling a different task when waiting for requested tokens, without stalling on a blocking receive primitive. Thus, the MPH can be considered as a first level of architectural support for adaptivity. The second level is represented by the insertion of the STM and the TMH, that have to take care of detecting faults and sending the migration data of the processes in the case of faulty processing elements. In Figs. 19 and 20, an estimation of the overhead due to the introduction of these modules is shown in terms of area occupation and maximum working frequency, respectively. The implementation results are obtained by means of the Xilinx tools during the protoyping phase.

It can be noticed that the overhead is not negligible. In terms of timing, the baseline architecture can be more than 25% faster than the NA featuring full support for fault-tolerance, especially due to the introduction of the MPH. During the design of the MPH architecture we tried to reduce as much as possible the latency related with message passing operations. This required the introduction of combinational logics which resulted in the mentioned frequency drop. A retiming of the control circuitry inside the MPH could be used to improve the achievable working frequency, at the price of an increment of the communication latency for each packet. The overhead in terms of used logic is also significant. Such overhead is mitigated when we consider the area of the entire tile, as shown in Fig. 21. In this case the area overhead in a tile with full support for adaptivity and fault-tolerance is almost 60% with respect to a tile instantiating the baseline NA. This overhead would be even smaller if we consider in the baseline area all the obstruction related to the memory modules, not accounted in the presented plot. It is also worth to notice that the baseline architecture cannot provide complete message-passing capabilities, thus it is not completely sufficient even in static message-passing systems.

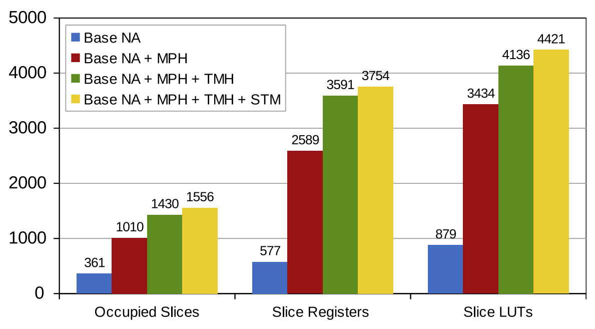

Fig. 19. Area occupation overhead in comparison to the baseline Network Adapter due to the support for system adaptivity and fault-tolerance.

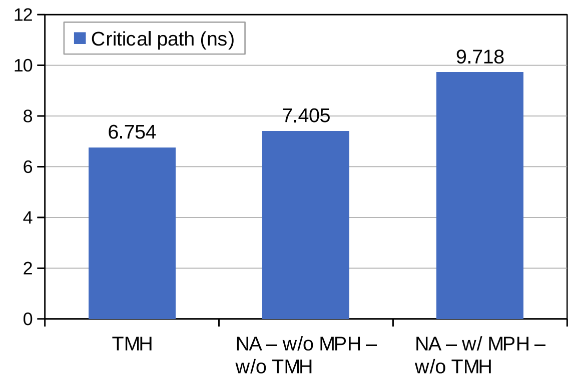

Fig. 20. Critical path length overhead related with support for system adaptivity and fault-tolerance.

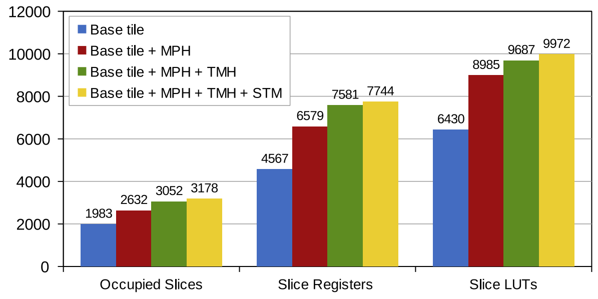

Fig. 21. Area occupation overhead in comparison to the baseline tile architecture due to the support for system adaptivity and fault-tolerance.

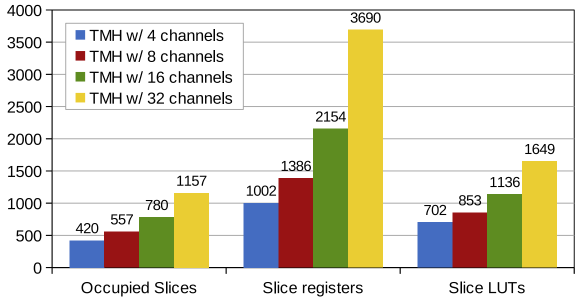

Fig. 22. Area overhead dependence on the supported number of channels.

Moreover, it is useful to point out that both the MPH and the TMH can be customized at design time, according to the communication graph of the target application, instantiating only the circuitry needed to control the required number of channels and tasks. As an example, we show how the TMH is customized for the H.264 and the MJPEG applications. In the first design case, the TMH has to support 4 tasks and 4 channels, requiring 35 registers to be instantiated. In the second, the circuitry must control 5 tasks and 8 channels, requiring 51 registers. The overhead comparison with the default configuration is shown in Fig. 22.

8. 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 in terms of execution time 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. However, the implemented architecture level support has a significant overhead that should be carefully assessed and limited when optimizing the design during the pre-product development phase.

Acknowledgements

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 from 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”.

References

[1] E. Cannella, L. Di Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer, A. Pimentel, Towards an ESL design framework for adaptive and fault-tolerant MPSoCs: MADNESS or not?, in: 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011, pp. 120–129.
[2] S. Verdoolaege, Handbook on Signal Processing Systems, Springer.
[3] S. Verdoolaege, H. Nikolov, T. Stefanov, pn: a tool for improved derivation of process networks, EURASIP J. Embed. Syst. 2007 (2007) 19.
[4] V. Nollet, D. Verkest, H. Corporaal, A safari through the MPSoC run-time management jungle, J. Signal Proces. Syst. 60 (2010) 251–268.
[5] G.M. Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, M. Robert, An adaptive message passing MPSoC frame-work, Int. J. Reconfigur. Comput 2009 (2009).
[6] S. Bertozzi, A. Acquaviva, D. Bertozzi, A. Poggiali, Supporting task migration in multi-processor systems-on-chip: a feasibility study, in: Proceedings of the Design, Automation and Test in Europe, DATE ’06, 2006, vol. 1, pp. 1–6.
[7] A. Acquaviva, A. Alimonda, S. Carta, M. Pittau, Assessing task migration impact on embedded soft real-time streaming multimedia applications, EURASIP J. Embed. Syst. 2008 (2008).
[2] C. Lee, H. Kim, H. woo Park, S. Kim, H. Oh, S. Ha, A task remapping technique for reliable multi-core embedded systems, in: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS), 2010, pp. 307–316.
[9] C.-L. Chou, R. Marculescu, Farm: fault-aware resource management in noc-based multiprocessor platforms, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, pp. 1–6.
[10] M. Dall’Osso, G. Biccari, L. Giovannini, D. Bertozzi, 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, ICCD’03, Washington, DC, USA, p. 536.
[11] E. Cannella, O. Derin, P. Meloni, G. Tuveri, T. Stefanov, Adaptivity support for MPSoCs based on process migration in polyhedral process networks, VLSI Des. 2012 (2012) 17.
[12] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, S. Ravi, Systematic software-based self-test for pipelined processors, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (2008) 1441–1453.
[13] N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, A. Gonzalez, MT-SBST: self-test optimization in multithreaded multicore architectures, in: IEEE International Test Conference (ITC), 2010, pp. 1–10.
[14] M. Scholzel, T. Koal, H. Vierhaus, An adaptive self-test routine for infield diagnosis of permanent faults in simple risc cores, in: IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012, pp. 312–317.

[15] M. Malek, A comparison connection assignment for diagnosis of multiprocessor systems, in: Proceedings of the 7th Annual Symposium on Computer Architecture, ISCA '80, ACM, New York, NY, USA, 1980, pp. 31–36.
[16] I. Koren, C.M. Krishna, Fault Tolerant Systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[17] O. Derin, E. Diken, L. Fiorin, A middleware approach to achieving fault-tolerance of Kahn process networks on networks-on-chips, Int. J. Reconfigur. Comput. 2011 (2011) 15.
[18] O. Derin, D. Kabakci, 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.
[19] D. Gizopoulos, Online periodic self-test scheduling for real-time processor-based systems dependability enhancement, IEEE Trans. Depend. Secure Comput. 6 (2009) 152–158.

Onur Derin received his B.Sc. in Electrical and Electronics Engineering from the Faculty of Engineering of Bogazici University (Turkey) in 2004 and his M.Sc. in Embedded Systems Design from University of Lugano – ALaRI (Switzerland) in 2006. He is currently a Ph.D. candidate at the Faculty of Informatics of the University of Lugano. His research interests include self-adaptation and fault-tolerance in the context of on-chip multiprocessors.

Emanuele Cannella received the B.Sc. and M.Sc. degrees in Electronic Engineering from University of Udine, Italy, in 2006 and 2008 respectively. He is now a Ph.D. candidate in Computer Science at Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands. His main research interests include runtime resource management in embedded multiprocessor platforms, mixed-criticality systems, and FPGA-based multiprocessor prototyping.

Giuseppe Tuveri received the B.Sc. and M.Sc. degrees in Electronic Engineering from University of Cagliari, Italy, in 2006 and 2009 respectively. He is part of EOLAB since March 2010, when he joined the Department of Electrical and Electronic Engineering of University of Cagliari, as a Ph.D. student. His main research interests include embedded operating systems, system adaptivity in embedded platforms, and FPGA-based multiprocessor prototyping.

Paolo Meloni is currently assistant professor at the Department of Electrical and Electronic Engineering (DIEE) in the University of Cagliari. In October 2007 he received a Ph.D. in Electronic Engineering and Computer Science, presenting the thesis "Design and optimization techniques for VLSI network on chip architectures". His research activity is mainly focused on the development of advanced digital systems, with special emphasis on the application-driven design of multi-core on-chip architectures. He is author of a significant record of international research papers and tutor of many bachelor and master students' thesis in Electronic Engineering. He is teaching the course of Embedded Systems at University of Cagliari and is currently part of the technical board and acting as work-package leader in the research projects ASAM (www.asam-project.org) and MADNESS (www.madness-project.org).

Todor Stefanov received the Dipl.Ing. and M.S. degrees in computer engineering from The Technical University of Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, The Netherlands, in 2004. From 1998 to May 2000, he was a Research and Development Engineer with Innovative Micro Systems, Ltd., Bulgaria. From June 2000 to August 2007, he was with the Leiden Institute of Advanced Computer Science, Leiden University, where he was a Research Assistant (Ph.D. student) and a PostDoc Researcher at the Leiden Embedded Research Center. From September 2007 to August 2008, he was a Senior Researcher at the Computer Engineering Lab, Delft University of Technology, The Netherlands. Since September 1, 2008, Todor Stefanov has been an Assistant Professor with the Leiden Institute of Advanced Computer Science, Leiden University where currently he is the head of the Leiden Embedded Research Center. Dr. Stefanov is a recipient of the 2009 IEEE TCAD DONALD O. PEDERSON BEST PAPER AWARD. His research interests include several aspects of embedded systems design, with particular emphasis on system-level design automation, multiprocessor systems-on-chip design, and hardware/software codesign. He serves on the organizational committees of several leading conferences and workshops in the field.

Leandro Fiorin obtained his Ph.D. from the Faculty of Informatics of the University of Lugano (USI), Switzerland, in 2012. He also received a Master of Engineering in Embedded System Design in 2004 from USI, and he holds a M.S. degree in Electronic Engineering, from University of Cagliari, Italy. He is currently research associate at USI – ALaRI. Previously he was also contract researcher at USI, working on networks-on-chip and embedded systems architectures. His research interests focus on fault tolerant and secure networks-on-chips and embedded systems, on-chip multiprocessors, reconfigurable systems. He is co-author of several scientific papers on networks-on-chip, design methodologies for systems-on-chip, embedded system security, and of two patents on networks-on-chip security.

Luigi Raffo (MSc Electronic Engineering (magna cum laude) in 1989, Ph.D. in Electronica and Computer Science in 1994, University of Genova, Italy) is full professor of Electronics at the Dept. Electrical and Electronic Engineering of the University of Cagliari, Italy. He is a teacher of electronics and system design courses. His main research field is the design of digital/analog devices and systems. In this field he has authored more than 80 international publications, and patents. He has been coordinator of EU, Italian Research Ministry, Italian Space Agency, industrial projects.

Mariagiovanna Sami is Professor, Digital Processing systems, at Politecnico di Milano. She holds an Electronics Engineer degree (Politecnico di Milano, 1966) and a Libera Docenza, Switching Theory and Computing (Italian Ministry for Education, 1971). Her research interests include various aspects of digital architecture design, with particular reference to defect and fault-tolerance of digital architectures, parallel architectures, low-power design and high-level synthesis. She is co-author and/or co-editor of several books and of over 200 technical papers. She has been Chairman of the Department of Electronics, Politecnico di Milano, and is at present Scientific Director of the ALaRI Institute, University of Lugano. Prof. Sami has been Editor-in-Chief of the Journal of Systems Architecture and member of the Board of Editors of IEEE Micro, IEEE Design and Test, IEEE Transactions of computers. She is a member of the Board of Editors of JETTA – Journal of Electronic Testing. She was General Chair or Program Chair for a number of international conferences chair; in 2000 she was General chair of IJCNN (the International Joint Conference on Computer Networks). She was also co-director of the NATO Advanced Study Institutes on VLSI Testing held in Como, Italy, in 1985 and on Hardware/Software co-design held in Tremezzo (Italy) in 1995.