Self-adaptivity of Applications on Network on Chip Multiprocessors

The Case of Fault-tolerant Kahn Process Networks

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Onur Derin

under the supervision of
Prof. Mariagiovanna Sami

May 2015

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera italiana, Lugano, Switzerland
Prof. Dr. Miroslaw Malek	Università della Svizzera italiana, Lugano, Switzerland
Prof. Fernando Pedone	Università della Svizzera italiana, Lugano, Switzerland
Prof. Dr. Peter Marwedel	Technische Universität Dortmund, Dortmund, Germany

Dissertation accepted on 19 May 2015

Research Advisor	PhD Program Director
Prof. Mariagiovanna Sami	**Prof. Igor Pivkin**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Onur Derin
Lugano, 19 May 2015

To my family

The way to succeed is to double
your failure rate.

Thomas J. Watson

Abstract

Technology scaling accompanied with higher operating frequencies and the ability to integrate more functionality in the same chip has been the driving force behind delivering higher performance computing systems at lower costs. Embedded computing systems, which have been riding the same wave of success, have evolved into complex architectures encompassing a high number of cores interconnected by an on-chip network (usually identified as Multiprocessor System-on-Chip). However these trends are hindered by issues that arise as technology scaling continues towards deep submicron scales. Firstly, growing complexity of these systems and the variability introduced by process technologies make it ever harder to perform a thorough optimization of the system at design time. Secondly, designers are faced with a reliability wall that emerges as age-related degradation reduces the lifetime of transistors, and as the probability of defects escaping post-manufacturing testing is increased.

In this thesis, we take on these challenges within the context of streaming applications running in network-on-chip based parallel (not necessarily homogeneous) systems-on-chip that adopt the no-remote memory access model. In particular, this thesis tackles two main problems: (1) fault-aware online task remapping, (2) application-level self-adaptation for quality management. For the former, by viewing fault tolerance as a self-adaptation aspect, we adopt a cross-layer approach that aims at graceful performance degradation by addressing permanent faults in processing elements mostly at system-level, in particular by exploiting redundancy available in multi-core platforms. We propose an optimal solution based on an integer linear programming formulation (suitable for design time adoption) as well as heuristic-based solutions to be used at run-time. We assess the impact of our approach on the lifetime reliability. We propose two recovery schemes based on a checkpoint-and-rollback and a rollforward technique. For the latter, we propose two variants of a monitor-controller-adapter loop that adapts application-level parameters to meet performance goals. We demonstrate not only that fault tolerance and self-adaptivity can be achieved in embedded platforms, but also that it can be done without incurring large overheads. In addressing these problems, we present techniques which have been realized (depending on their characteristics) in the form of a design tool, a run-time library or a hardware core to be added to the basic architecture.

Acknowledgements

I am indebted to numerous people for their support in the creation of this thesis. First and foremost, this work would not be possible if it was not for the hospitable environment offered by ALaRI and University of Lugano. For that I am forever grateful to the administration, in particular, to Prof. Mariagiovanna Sami, Prof. Miroslaw Malek, Umberto Bondi and the late Prof. Luigi Dadda. Mariagiovanna, as my research advisor, has provided me with continuous support, guidance and patience. Not only as a scientist but as a person, her example sets a high standard to follow for the rest of my life.

I feel privileged to have taken part in the MADNESS Project and I would like to thank all of its members, in particular Prof. Luigi Raffo, Paolo Meloni, Giuseppe Tuveri and Sebastiano Pomata (University of Cagliari) for providing the baseline platform upon which our implementation is built; Prof. Todor Stefanov and Emanuele Cannella (University of Leiden) for hosting me during my research visit and HiPEAC Network of Excellence for providing me a collaboration grant. I would also like to thank the members of my dissertation committee whose insightful comments helped me to improve this work.

While carrying out the work presented in this thesis, I had the opportunity to tutor some master of science students enabling me to investigate a variety of research problems and to offload some of the required engineering effort. Throughout the years our relationships have evolved into dear friendships. I hope I have been as impactful in their lives as they have been in mine. In particular, I would like to thank Deniz Kabakcı for the implementation of remapping heuristics in C; Karim M. A. Ali for coding of the TMH in VHDL; Poorna C. S. Alamanda for helping with tedious roll-forward recovery experiments; Prasanth K. Ramankutty for the implementation of the fuzzy controller and carrying out extensive experiments; Mariano Perna and Enea Affini for suffering the bugs of the platform and helping in their identification; Erkan Diken for exploring self-adaptation with the SACRE-Noxim integration; Katarina Balać and Aleksandra Jovanović for exploring self-adaptation in the GStreamer framework; and lastly Erick Amador, Amrit Panda, Yücel Şahin and Sherif El-Wafa for helping me ex-

plore other research areas towards the beginning of my studies.

I am also thankful to all of my colleagues at ALaRI, in no particular order, Leandro, Jelena, Antonio, Alberto, Giovanni, Francesco, Daniela, Katarina, Slobodan, Igor, Rami, Yum, Elisa, Janine, Carola, Luca and Marco for their support, availability and friendship. Life in Lugano has been even sunnier for me thanks to some of my friends, of whom I can list only a few: Cumhur, Ali, Korman, Gamze, Sertuğ, Burcu, Seda, Oktay and Caner. Without their moral support, these years would have been more difficult and less joyful.

Finally, the process leading to the completion of this thesis was as hard for my wife, Rana, as it was for me. I cannot thank her enough for being by my side. My deepest gratitude is to my parents who have lived through this world by making one not from zero but from minus one.

Contents

Figures

Tables

List of Abbreviations

API application programming interface.

ASIP Application Specific Instruction Processor.

BDD binary decision diagram.

BER backward error recovery.

BIST built-in self-test.

BR bit-rate.

CMOS complementary metal-oxide-semiconductor.

CoG Center of Gravity.

CPU central processing unit.

CRC cyclic redundancy check.

CRR checkpointing-and-rollback recovery.

DMA direct memory access.

DMR double modular redundancy.

DSE design space exploration.

DSP digital signal processor.

EM electro-migration.

FER forward error recovery.

FIFO first-in first-out.

FPGA Field-Programmable Gate Array.

FR frame-rate.

GPU graphics processing unit.

HCI hot carrier injection.

HW hardware.

ILP integer linear programming.

IP intellectual property.

JPEG Joint Photographic Experts Group.

KPN Kahn Process Networks.

LNMS Localized NMS.

M-JPEG Motion JPEG.

MCA monitor-controller-adapter.

MCA-EB event-based MCA using blocking channels.

MCA-EI event-based MCA using interrupts.

MoC model of computation.

MPEG Moving Picture Experts Group.

MPH DMA message-passing handler.

MPI Message Passing Interface.

MPSoC Multiprocessor System-on-Chip.

MTOS multi-threaded operating system.

MTTF mean-time-to-failure.

NA Network Adapter.

NBTI negative bias temperature instability.

NI network interface.

NMR N-modular redundancy.

NMS nonidentical multiprocessor scheduling.

NoC Network-on-Chip.

NORMA no-remote memory access.

NUMA non-uniform memory access.

OS operating system.

OTR fault-aware online task remapping.

PE processing element.

PPN Polyhedral Process Networks.

QoS quality of service.

RFR roll-forward recovery.

RISC Reduced Instruction Set Computing.

RM run-time manager.

RTE run-time environment.

SDC silent data corruption.

SDF Synchronous Data Flow.

SM stress migration.

SoC System-on-Chip.

STM Self-testing Module.

SW software.

TDDB time dependent dielectric breakdown.

TMH task migration hardware.

TMR triple modular redundancy.

Chapter 1

Introduction

Guided by Moore's Law, technology scaling accompanied with higher operating frequencies has been the driving force behind delivering higher performance computing systems at lower costs. However the past decade has witnessed the rise of some barriers, namely the power wall, the memory wall and the reliability wall, which threaten the rule of Moore's Law in the semiconductor industry [Borkar et al., 2007]. High performance embedded systems are also affected by these barriers as power consumption and reliability are even more important concerns for such systems. Power and memory wall have been regarded as more imminent threats and new architectural design trends have emerged in an effort to avert them.

Thanks to the advances in micro- and nano-electronic technologies, enabling the integration of billions of transistors in the same on-chip die, the next generation of embedded platforms will be composed of a high number of heterogeneous processing and storage elements; performances are increased by task-level parallelism, distributing the tasks on a multiplicity of (relatively simple) processors rather than by going for single, highly complex units running at very high frequency.

The initial trend was designing multi-core chips, usually in the form of a symmetric Multiprocessor System-on-Chip (MPSoC), with a limited number of nodes consisting of CPU and L1 cache interconnected by simple bus connections and capable in turn of becoming nodes in larger multiprocessors. However, as the number of components in these systems increases, communication becomes a bottleneck and it hinders the predictability of the metrics of the final system.

Networks-on-Chip (NoCs) [Dally and Towles, 2001] appeared as a design paradigm allowing to overcome the efficiency and technology problems related to traditional solutions for inter-core communication. NoCs, among the other

advantages, improve scalability, available bandwidth, and power efficiency of complex MPSoCs, usually by implementing a packet-switched communication among the cores [Benini and De Micheli, 2002]. The 48-core *Single Cloud Computer* [Howard et al., 2011] and the 80-core *Teraflops Research Chip* [Vangal et al., 2008] processors from Intel; the 64-core *Tile64* processor [Wentzlaff et al., 2007] from Tilera; and the 64-core *Epiphany* [Adapteva Inc., 2014] processor are few of the commercial NoC-based platforms exemplifying these trends.

Memory organization is also an important design choice. Shared memory architectures are less scalable than distributed-memory solutions. Although NoC-based platforms can support shared memory access, memory coherence protocols induce an overhead in the communication network rendering the gain from additional cores useless. There are two design paradigms for non-symmetric memory organization in NoCs: non-uniform memory access (NUMA) and no-remote memory access (NORMA). In NUMA, all cores share one logical address space, but this address space is physically partitioned so that each node has a local segment of the memory space; communication takes place simply by accessing shared memory locations. In NORMA, each core has its private local memory: this solution is suited for programming models based on message-passing [Carara et al., 2007].

This thesis takes on the challenges described in the next section in accordance with the aforementioned trends.

1.1 Motivation

1.1.1 The need for self-adaptation

Embedded systems are often subject to stringent non-functional goals such as high computational performance, low power consumption, restrictions on memory dimensions, low chip area and high dependability. Goals are specified by programmers or users by stating that a certain metric should be above or below a threshold value, or that it should be minimized or maximized. In classical design space exploration (DSE), systems are designed by evaluating the design alternatives in terms of the relevant metrics and by picking one that satisfies the goals, possibly with a trade-off between conflicting objectives.

Satisfying the non-functional requirements imposed by the application designer on systems with increasing complexity of the underlying architectures is a fundamental challenge. Design activities are mainly hindered by the difficulty in analyzing and estimating the performance metrics of the system. Design time

choices may be less than satisfactory when confronted with run-time processing, due to the complexity of the design space. Moreover, some run-time factors, which are only known during operation, may cause a performance that is different than the expected one. Variability issues of transistors at deep-submicron scale due to both manufacturing and aging effects show major performance and power consumption variability in the final system and thus they can no longer be overlooked [Borkar et al., 2007]. In order to deal with this problem, designers more and more often resort to self-adaptation based techniques [Kramer and Magee, 2007; Józwiak, 2006]. Self-adaptivity is the ability of the system to adapt itself dynamically to achieve its goals. As the complexity of the components increases and their integration becomes difficult, some of the decisions that are taken at design time are deferred to run-time. From that perspective, one can view self-adaptation as the ability of the system to switch at run-time from one design point to another in the design space with the help of a run-time design space exploration logic.

Self-adaptive systems are able to react when the actual operating conditions of the system differ from the design-time assumptions such as the workload, the internal and external conditions and the non-functional goals. For example, a portable device may be frequently moved from an office environment (where power and network plugs are available) to an external environment (where the device can only be battery operated and the network may be available in different wired or wireless forms). In this case the behavior of the system needs to be adapted to the new conditions (e.g., to reduce power consumption). Again, a video encoding application running on such a device may be required to work at higher resolution (i.e. different workload), at higher frame rate (i.e. different goal) and alongside a new application launched on the system (i.e. different internal condition).

There are certain challenges to be tackled when designing self-adaptive systems. A general concern when making the system monitorable and adaptable is the overhead introduced in the metrics of interest such as time, area and power. The benefits of adaptation can easily be offset by a large overhead. There are two types of overhead. The first type, which can be called *steady state overhead*, is the overhead experienced simply due to the additional hardware or software for enabling monitoring and adaptation capabilities. It is present even when there are no ongoing adaptations. This overhead should be minimized because it is afforded at all times. The second type, which can be called *transient overhead*, is the overhead experienced while an adaptation is taking place. The major sources of this overhead are the adaptation algorithm of the controller and the execution of an adaptation. If the system is expected to have frequent adaptations, then

care must be taken to minimize this type of overhead.

Separation of concerns is a key feature for self-adaptive systems. However the realization of this principle is quite challenging for several reasons. It emphasizes that the application programmer should be involved as little as possible in making the system self-adaptive. Although it may be possible to realize this for adaptations at the run-time environment and hardware levels because of the clear interface between the application and the execution platform, it is a more difficult task for application level adaptations. Intrinsic application knowledge by the application programmer is required in order to expose the feasible adaptations in the application. Automatic inference of such adaptations would be very difficult, if possible at all. Depending on the adaptation goal, another difficulty is in the inference of what to monitor and how to monitor it without involving the programmer. There is a semantic gap to be bridged between the given goal and the application. Monitoring involves choosing the correct program variables and operating on them in order to calculate the actual metric that corresponds to the goal. Another issue with the separation of concerns principle is that it is likely to conflict with the low overhead goal mentioned previously. Decreasing the amount of load on the programmer would lead to increasing the amount of work that the self-adaptation logic has to do, thus leading to greater overhead due its complexity. Last but not least, the behaviour of the adaptation controller is application-dependent. Machine learning algorithms can be used to obtain the application knowledge, particularly the relation between the goal and the adaptations, but this would result in a complex control logic with a larger overhead. Alternatively, the required application knowledge can be provided to the controller by the application programmer.

Another fundamental challenge for system-wide self-adaptivity is presented by the management of the adaptations. The systems are usually faced with multiple goals to be satisfied at run-time such as a desired throughput, low power consumption, high dependability. Satisfying all the goals by controlling various possible adaptation options is a difficult task. Changing the set of goal types would require a complete or partial re-design of the controller. Possible solutions to solve this problem are automated controller synthesis or designing generic controllers.

1.1.2 The need for fault tolerance

Over the years semiconductor industry has benefited greatly from the improvements in process technologies that enabled scaling down of CMOS transistors. Smaller transistors led to better circuit performance due to higher frequencies.

As the area of the chip got smaller, fabrication costs per chip were also reduced. Moreover, the ability to pack more transistors in a smaller area enabled the integration of many more functions in a chip, leading also to performance improvements. These trends, referred to as Moore's Law, are threatened by power and reliability concerns in the deep sub-micron scale. As highlighted by the International Technology Roadmap for Semiconductors [ITRS, 2009], these concerns have to be addressed to continue harvesting the benefits of *More-Moore*.

A hardware fault is caused by an underlying physical defect in hardware. An error is the external manifestation of a fault. Regarding their duration, hardware faults have been classified as permanent (hard), transient (soft) and intermittent [Koren and Krishna, 2007]. Permanent faults cause persistent component malfunction, while transient faults last only for some time. Intermittent faults are those that make a component oscillate permanently between malfunctioning and correct functioning. We limit the scope of this thesis to permanent faults as they determine the lifetime reliability of a system. For systems that are prone to radiation effects, which are the major cause of transient errors, techniques presented in this thesis should be accompanied with techniques that handle such errors.

Permanent faults can be extrinsic or intrinsic. The former are caused by manufacturing defects and can be mostly detected by post-manufacturing tests such as burn-in. The latter are caused by wear-out and manifest during operation. The failure rate of electronic systems usually adhere to the Weibull distribution, also known as the bathtub curve [Koren and Krishna, 2007]. Extrinsic faults exhibit as high infant mortality rate, while intrinsic faults account for the increasing failure rates in the wear-out phase. In between the two lies the useful lifetime phase with a lower constant failure rate.

Considering an individual transistor, the fundamental mechanisms leading to intrinsic permanent failures are electro-migration (EM), stress migration (SM), time dependent dielectric breakdown (TDDB), negative bias temperature instability (NBTI) and hot carrier injection (HCI) [Renesas, 2013]. EM is caused by the movement of metal atoms in the interconnect with the momenta of the electrons, leading eventually to an open or a short circuit. SM stems from the thermal stress due to different thermal expansion coefficients of the interconnect metal and the underlying film, leading possibly to a rupture in the interconnect. In TDDB, a conducting path gradually forms between the gate electrode and the silicon substrate due to the traps created inside the dielectric by the electric field between the gate and the substrate. NBTI is caused by the formation of positive charge inside the dielectric due to high negative bias at the gate (with respect to source and drain) and high temperature, leading to the degradation of PMOS transistor's threshold voltage and thus performance. Similarly, NMOS transis-

tors suffer PBTI. HCI is caused by high electric field around the drain due to high supply voltage, leading to the injection of carriers (electrons or holes) inside the dielectric, thus degrading transistor's performance.

These failure mechanisms are affected strongly by the feature sizes of the transistor and in particular by the electric field at the gate, the temperature and the supply voltage. Although ideal scaling rules of transistors aim at keeping the electric field constant, in practice the supply voltage has been kept above ideally scaled values for performance concerns. This has two implications, firstly, the power densities have been increasing, and secondly, the ratio of leakage power with respect to switching power has been increasing with each technology node. As a result, chip temperatures have also increased, deteriorating the lifetime reliability of transistors [Borkar, 2005].

Moving from transistor to chip level, as the probabilty of failure in individual transistors increases, the probabilty of a failing transistor in a chip with billions of them becomes a more threatening issue for the lifetime of the system. While memories are often protected with error detecting and correcting codes, components consisting majorly of logic such as the processing elements (PEs) are more vulnerable to hard failures. Srinivasan [2006] estimates a five-fold decrease in a processor's lifetime as it is scaled from 180nm to 65 nm technology and expects this decrease to occur at higher rates with further scaling.

Another concern calling for fault tolerance against operational faults is due to the limitations of post-manufacturing tests to catch early defects. In particular, the effectiveness of the burn-in process, which subjects the final product to testing at higher voltages and higher temperatures to accelarate aging, is hampered by device scaling due to the prohibitive (exponential) increases in the gate leakage [Borkar et al., 2007] and by decreases in supply voltages (resulting in exponentially longer burn-in times) [ITRS, 2009].

As a result, designers are faced with the challenge of handling faults at runtime by detection, isolation and recovery without introducing prohibitive performance and cost overheads. Although reliability improvements in process technologies are continuously sought, a permanent solution to this problem requires new design techniques to be developed. Constraints presented by embedded systems design (e.g., low cost and low power consumption) make traditional approaches involving massive redundancy hardly adoptable. Moreover assessing the reliability of the system, consisting of the application and the platform it runs on, only after the platform is available puts the product development at risk with regard to time-to-market requirements. Design flows that encompass a loop in order to incorporate a posteriori measures for meeting reliability constraints are rendered obsolete. Therefore design techniques embracing con-

tinued availability in the presence of faults must be embedded into the normal design flow. This brings in a new challenge on the part of the designers of these techniques. In order to develop such techniques, low level fault models strongly tied to the technology should be abstracted in the form of functional-level error models. Since the exploitation of knowledge at higher levels of abstraction (e.g., regarding the application, the error or the system) creates a potential for developing fault tolerance techniques with lower performance and area costs; there has been a trend to move from physical and logical to micro-architectural, architectural and software-based solutions. In fact, these techniques employed at several levels should work hand in hand to guarantee reliable computation from unreliable components. This design paradigm which distributes the responsibility for tolerating errors, device variation and aging across the system stack is known as cross-layer resilient design [Mitra et al., 2010; Carter et al., 2010; DeHon et al., 2010]. By viewing fault tolerance as a self-adaptation aspect, in this thesis we adopt a cross-layer approach that aims at graceful performance degradation by addressing fault tolerance mostly at system-level, in particular by exploiting redundancy available in multi-core platforms. By system-level, we refer to techniques implemented in software both at application and run-time environment levels and not purely in hardware.

1.2 Research framework

The work presented in this thesis has been carried out within two European projects. Initially it benefited from our partial involvement in the AETHER EU FP6 Project (No. IST-027611) – Self-Adaptive Embedded Technologies for Pervasive Computing Architectures. As it will be mentioned in section 2.1, the common design pattern for self-adaptivity is the monitor-controller-adapter (MCA) loop. Self-adaptive systems can be characterized with respect to what can be monitored, controlled and adapted. As a part of our work within the AETHER Project on modeling system-wide self-adaptivity in embedded systems, we proposed a generic model consisting of three levels, which are application, run-time environment (RTE) and hardware as shown in figure 1.1 [Derin et al., 2009]. In each of these levels, there are parameters to be monitored and adapted. To exemplify each level, at the application level, some application parameters or variant implementations of some functions can be adapted; some performance goals such as throughput and delay can be monitored. At the RTE level, the mapping of tasks onto the resources as well as the scheduling policy of tasks on a given resource can be adapted; resource utilization, goal achievements per application

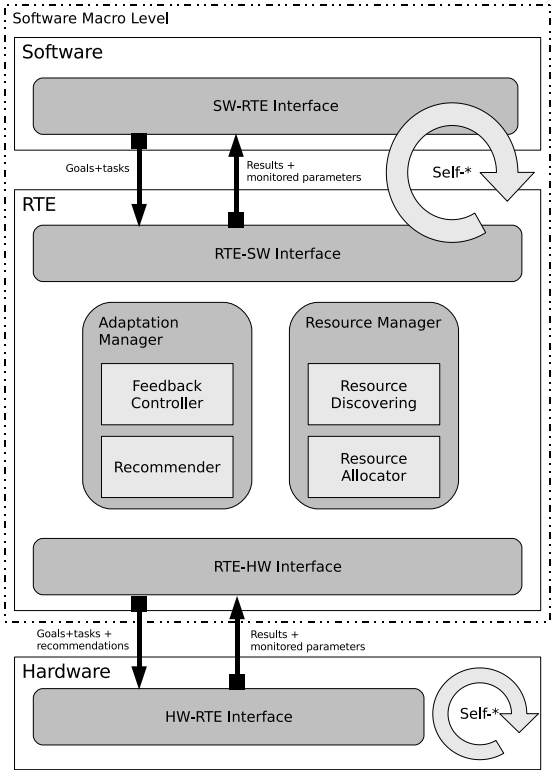


Figure 1.1. Model of a self-adaptive system

can be monitored. At the hardware level, the clock frequency and voltage levels can be adapted; power consumption and temperature can be monitored. A goal is an expression containing boolean and arithmetic operators, which evaluates to true or false to denote whether the goal is satisfied or not. The variables in a goal expression come from the monitors. Controllers provide commands to the adaptors to change the system configuration. A goal is assigned to a controller at a specific level. For example, a performance goal for an application is associated with the application level, whereas a power consumption goal can be assigned to the RTE or hardware levels. Assigning the goals to the level where there are a relevant adaptor and monitor is a wise choice; otherwise, some overhead to monitor and/or adapt parameters of another level would be involved. Moreover, this is likely to violate the seperation of concerns principle.

Our AETHER work has also produced some ideas for enabling application-level self-adaptation capabilities for process networks [Derin and Ferrante, 2009] and middleware support for Kahn Process Networks (KPN) on NoC-based plat-

forms [Derin and Diken, 2010] with consideration of adaptability and fault-tolerance [Derin, Diken and Fiorin, 2011]. However, it is the MADNESS Project that allowed us to move from simulated systems to prototyped solutions.

Most of the work presented in this thesis has been done within the MADNESS EU FP7 Project (No. ICT-248424) – Methods for predictAble Design of heterogeneous Embedded Systems with adaptivity and reliability Support – in coordination with the project partners. Both reference architecture and benchmark application(s) were chosen in agreement with the whole collaboration.

The project aims at the definition of innovative system-level design methodologies for embedded MPSoCs, extending the classic concept of design space exploration in multi-application domains to cope with high heterogeneity, technology scaling and system reliability. The main goal of the project is to provide a framework able to guide designers and researchers to the optimal composition of embedded MPSoC architectures, according to the requirements and the features of a given target application field. The proposed strategies tackle the new challenges, related to both architecture and design methodologies, arising with the technology scaling, the system reliability and the ever-growing computational needs of modern applications.

1.2.1 KPN and PPN as the model of computation

The design of embedded systems, unlike general computing systems, involve not only functional goals but also non-functional ones that dictate performance and/or resource usage goals. As a consequence, design flows entail the adoption of a model of computation (MoC). The behavioral specification of the system can be expressed with the given MoC. But, most importantly, the MoC enables the analyzability of the system with regard to the given non-functional goals. Depending on the target application domain, system-level synthesis approaches adopt a variety of MoCs based on finite state machines or process networks [Gerstlauer et al., 2009].

A fundamental design decision taken at the beginning of the MADNESS Project by the partners was the adoption of a streaming application model based on KPN [Kahn, 1974], in particular its variant named Polyhedral Process Networks (PPN). Although KPN was introduced by Kahn in 1974, it was only after year 2000 that it re-gained attention with the emergence of the increasingly parallel platforms needed by high performance embedded applications, which constitute a target domain for the MADNESS Project. KPN and PPN models are based on the idea of organizing an application into streams (*channels*) and computational blocks (interchangeably referred to as *tasks* or *processes*); channels represent the

flow of data, while tasks represent operations on a stream of data. Further background on KPN and PPN is provided in sections 2.3 and 2.4, respectively.

KPN and PPN present themselves as an acceptable trade-off point between abstraction level and efficiency versus flexibility and generality. The favorable features of KPN and PPN, which enable the work presented in this thesis, are discussed in the following.

Generality: It is capable of representing many signal and media processing applications, which occupy a large percentage of the consumer electronics in the market. Some KPN application examples that can be found in the literature are image/video processing (JPEG [de Kock, 2002], M-JPEG [Lieverse et al., 2001], MPEG-2 [van der Wolf et al., 1999], H.264 [Zrida et al., 2008; Nikolov et al., 2009; Vrba et al., 2009], Sobel edge detection, 2D-DWT [Verdoolaege et al., 2007]), sound processing (ADPCM [Ceng et al., 2008]), telecommunication (GSM [Castrillon et al., 2010], software-defined radio [Castrillon et al., 2011]), security (AES [Vrba et al., 2009]) and scientific computation (QR decomposition [Stefanov et al., 2002]). It should further be noted that, a recent work by Thies and Amarasinghe [2010] has shown that most of the streaming applications can be specified using the Synchronous Data Flow (SDF) model [Lee and Messerschmitt, 1987a]. KPN and PPN models are more expressive than SDF, thus it can as well be used effectively to model all streaming applications that can be specified via SDF. Moreover, KPN can be used to express applications from other domains, thanks to its Turing-completeness, albeit they are likely to be less efficient.

Abstraction level: Being untimed MoCs based on asynchronous message passing, KPN and PPN guarantee functional correctness independent of timing. The programmer is not directly involved with concurrency management. KPN and PPN tasks can be implemented in any programming language. The only restriction imposed on the tasks is the communication interface consisting of the read and write operations. Most importantly, the performance of the system given the mapping of a KPN application on a platform is estimable with an analytical model. Moreover, in case of PPN, as the FIFOs are bounded, the memory requirement of the system is also estimable.

Efficiency: KPN and PPN enable the exploitation of the parallel processing power available on the NoC-based multiprocessor platforms. Organizing the computation as parallel tasks and overlaping the computation with communication using FIFOs allows an efficient implementation when combined also with a mapping exploration design phase. Since the communication between the tasks is exposed explicitly, it makes KPN and PPN very suitable for message-passing platforms such as the NoC-based multi-processor platforms with no remote mem-

ory access. This enables estimating the amount of communication in the system.

Flexibility: The techniques to be developed in order to address self-adaptation and fault-tolerance challenges depends on the adopted MoC. KPN/PPN makes it possible to adapt the application without excessive effort and overhead. A fundamental property which facilitates this is that KPN/PPN tasks do not require a global scheduler and can synchronize simply by blocking read operations. Such a flexibility enables remapping of tasks at run-time. Secondly PPN tasks have a special execution point (i.e., the beginning of their outmost loop bodies) where they posses a small state, thus enabling efficient remapping of the tasks onto new processing nodes.

1.3 Dissertation contributions

In the view of the aforementioned challenges, this thesis presents novel contributions for achieving fault-tolerant and self-adaptive applications which are modelled as process networks and run on top of NORMA-based NoC platforms.

The self-adaptation model provides a comprehensive basis for the work presented in this dissertation. The two concrete self-adaptation problems shown in table 1.1 are derived based on this model. Firstly, the fault tolerance goal at the RTE level is addressed by viewing the problem of fault tolerant execution of KPNs on NoCs as a self-adaptation problem. Fault detection corresponds to monitoring; the remapping decision (chapter 4) corresponds to the adaptation control; and the fault recovery via task migration (chapter 5) corresponds to the adapter. Secondly, quality goals at the application level are addressed by viewing the adaptation of the parameters of KPN tasks as a self-adaptation problem as described in chapter 6. The quality attributes that are defined in [Derin et al., 2009], namely control quality, steady-state overhead and separation of concerns, have been used as the assessment criteria for the presented solutions to these two self-adaptation problems.

Figure 1.2 shows a pictorial view of the concrete self-adaptive and fault-tolerant system that has been realized in this thesis where the contributions are marked with their corresponding chapter numbers.

The main contributions of the dissertation are as follows:

- **Fault-aware online task remapping (OTR):** Firstly, we propose an integer linear programming (ILP) based method for finding the Pareto-optimal mappings and remappings of KPN applications onto NoC-based platforms with consideration of computation and communication objectives. Although optimal remappings can be computed offline for a given application and

Table 1.1. Addressed self-adaptation problems

Problem	**Fault-aware online remapping**	**Application-level self-adaptation for quality management**
Goal space	fault tolerance	performance
Monitorable space	permanent faults in PEs	throughput
Adaptation space	task mapping	task parameters
Controller	remapping algorithms	quality controller
Control quality	performance degradation, fault recovery time	mean absolute error, rise/fall time
Other quality criteria	steady-state overhead, separation of concerns	steady-state overhead, separation of concerns

stored in system's memory to be used when a fault is encountered, it becomes inapplicable with increasing number of tasks and processing nodes, which often have limited memory resources. For this reason, we adopt an online approach and propose heuristics for the problem of remapping tasks when run-time faults are encountered. We assess the quality of the heuristics by comparing them with the optimal solutions found by the ILP-based method. We also evaluate the computation times of the heuristics on the actual platform.

We also present an analytical model for the calculation of the mean-time-to-failure (MTTF) as a reliability metric when running KPN applications on NoC-based platforms. We operate on an abstract, high-level model rather than on detailed lower-level solutions, thus achieving a model that remains valid for multiple underlying implementations. We investigate transformations of the KPN task graph that make it more reliable and allow fault detection and/or fault masking. We provide a comparison between the fault-aware online remapping technique and application-level N-modular redundancy in terms of their reliability and performance overhead.

- **Recovery support in the fault-aware run-time environment (CRR and RFR):** We realize the fault-aware online task remapping concept on a NoC-based platform in the form of a self-adaptive run-time environment consisting of a task-aware middleware, fault recovery support via task migration and a run-time manager. We propose roll-forward recovery (RFR) and checkpointing-and-rollback recovery (CRR) schemes. For each scheme, the recovery support involves modifications to the process template, run-time support in the form of interrupt handlers and addition of a task migration

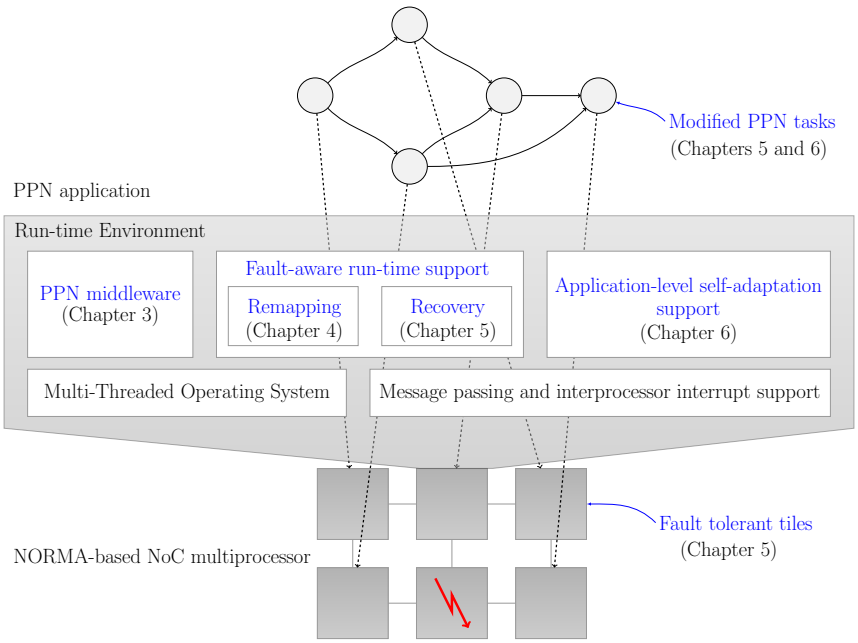


Figure 1.2. Overview of the proposed self-adaptive and fault-tolerant system

hardware module to the tile architectural template. Both recovery schemes have been implemented and compared in terms of their overheads.

- **Application-level self-adaptation for quality management (MCA-EB and MCA-EI):** We realize quality management support as a part of the self-adaptive run-time environment consisting of monitoring tasks, adaptable tasks and controller tasks based on fuzzy logic. Two schemes are proposed based on an MCA loop that interacts with the application, firstly, via blocking channel semantics, and secondly, via interrupting messages. The two schemes have been implemented and compared in terms of their overheads.

1.4 Organization of the dissertation

The remainder of this thesis is organized as follows:

Chapter 2 presents some background topics, in particular, regarding self-adaptive systems, MPSoC programming models, Kahn Process Networks and Polyhedral Process Networks. Then, an overview of the related work on each of the individual parts addressed in the dissertation is provided, in particular, regarding KPN frameworks in MPSoCs, mapping of applications onto NoCs, fault

tolerance in embedded systems with a focus on system-level approaches and finally application-level self-adaptation.

Chapter 3 presents firstly the preliminaries of this thesis by describing the baseline platform which forms the starting point of the implementation work. Then we present the developed middleware which enables the fault tolerance and self-adaptivity techniques that have been implemented in the following chapters. Lastly, the adopted fault tolerance approach is described at large.

Chapter 4 presents the fault-aware online task remapping approach. Firstly, the ILP formulation for the optimal solution of the mapping problem is described based on an analytical model for throughput and communication cost. Then it is extended for the remapping problem and a set of heuristics for online task remapping are explained. After that, we propose an analytical model for estimating the lifetime reliability of a system that adopts the fault-aware online task remapping technique. N-modular redundancy technique is also considered for comparison purposes. The chapter concludes with several case studies and results obtained both analytically and experimentally on the actual platform.

Chapter 5 presents the two techniques proposed for fault recovery based on fine-grained checkpointing-and-rollback and roll-forward. Each section details the required changes at the application, RTE and hardware layers. We present results obtained experimentally on the platform and compare the two techniques in terms of their overhead in time and area.

Chapter 6 presents the two proposed approaches for a self-adaptive framework with implementation details of monitoring, controlling and adaptive tasks. We give details on how a self-adaptive M-JPEG encoder case study is built using the two frameworks. Then we provide the results of the case study with a comparison of the two approaches in terms of the steady-state performance overhead and quality of the control.

Finally, chapter 7 concludes with a summary of achieved results and discusses possible extension points for future work.

Chapter 2

Background and Related Work

In the following sections, firstly, we provide general background on self-adaptive systems in section 2.1. Then, some background information about MPSoC programming models is given in section 2.2. The two models of computation of interest to us, namely KPN and PPN, are described in section 2.3 and 2.4, respectively. Previous work on KPN realizations in MPSoCs is discussed in section 2.5. Mapping of applications onto NoC platforms is overviewed in section 2.6 followed by task migration in section 2.7. Then, background information on fault tolerance is given together with the related work on fault tolerance and lifetime reliability approaches in embedded systems in section 2.8. The chapter ends with the related work on application-level self-adaptation for quality management in section 2.9.

2.1 Self-adaptive systems

As the most prevalent technique for self-adaptation, we see the use of the monitor-control-adapt paradigm, also referred to as the autonomic control loop, the monitor-analyze-plan-execute loop or the monitor-analyze-decide-act loop [Dobson et al., 2010]. The main idea is monitoring internal and/or external conditions and adapting the system according to a control logic in order to satisfy the goals. In an effort to classify the existing solutions, we identify four main design decisions:

- *Adaptation coverage* is defined by the parts of the system affected by adaptations. It may consist of hardware, software, part of a distributed system (through adaptive middleware) or any combination of them.
- *Separation of concerns* is a design principle that decouples the functionality of the system from the implementation of its self-adaptation capability.

- *Adaptation management* is the decision making process on the evolution of the system, in other words, the adaptation control logic.
- *Adaptation requirements specification* is the form of describing the non-functional requirements of the system.

In the remaining part of this section we provide some examples of related work on self-adaptivity, classified according to their most prominent characteristics in the view of the above-listed criteria. A more complete list of related work on self-adaptive systems can be found in [Salehie and Tahvildari, 2009; Cheng et al., 2009; Lemos et al., 2013]. Similar concepts has been studied also under other research areas such as autonomic computing [Huebscher and McCann, 2008] or organic computing (a term coined by the German Organic Computing Initiative) [Schmeck, 2005].

2.1.1 Adaptation coverage

A number of earlier studies address self-adaptivity in software; the simplest approach adopted is to manage adaptation in the application code. Although this approach enables the development of ad hoc solutions for specific adaptation problems, it is clearly not flexible enough to support a wide range of adaptations. The use of an *architecture-based approach* eases self-adaptivity: the system is viewed as a composition of concurrent *components* interconnected by *connectors*. A comprehensive adaptation methodology is presented in [Oreizy et al., 1999]. The authors propose an evolution and adaptation management infrastructure. The evolution management process adapts the architecture and the topology of the components and of the system; the adaptation management process gathers information from the operating environment, evaluates the observations with respect to the system requirements, plans and deploys adaptation changes. Moreover the need of composable components is emphasized. Another work describing a component-based architectural approach is presented in [Garlan et al., 2004]. The authors propose a framework that is both reusable, to cope with a large set of systems, and that supports mechanisms to specialize the infrastructure for specific cases. To achieve such objectives, the framework is divided into two logical parts: an adaptation infrastructure and a system specific adaptation model. The former provides common functionality that is reusable across different self-adaptive systems; the latter is specific to a certain system and it is used to tailor the entire framework for it. Similarly, Geihs et al. [2009] propose a comprehensive solution for the development and operation of context-aware, self-adaptive applications. The main contributions of this work are (a) a

sophisticated middleware that supports the dynamic adaptation of component-based applications, and (b) an innovative model-driven development methodology based on abstract adaptation models and corresponding model-to-code transformations. In [Balasubramaniam et al., 2004] an architecture description language named ArchWare is modified to support self-adaptation. Feedback obtained by means of software probes is used to control software self-adaptations.

A formal approach to the design of adaptive software is introduced in [Zhang and Cheng, 2006]. In particular, the adaptation is conceived as a state transition from a source program to a target program inside a suitable set of adaptation states. Each adaptive software is represented by a state machine, where each state exhibits a different behavior and operates in a certain domain. To guarantee system integrity and consistency, local and global properties (requirements, constraints, and invariants) that should be satisfied by an adaptive program for every state change are introduced.

In [Gjørven et al., 2006] a *mirror-based reflection* approach for self-adaptivity is proposed. By definition, a reflective system is able to perform computations about itself; moreover, it provides introspection and control through a reflective interface. By applying this reflective mechanism to software components, the middleware can perform self-adaptation by using the reflective interface of each component. Adaptation behavior, architecture and implementation of a component can be specialized to fit a specific context by annotating each implementation with quality of service metrics. Therefore, the middleware uses such quality of service metrics to trigger the adaptation of components.

Some studies related to hardware self-adaptation have also been proposed. Self-adaptation at the hardware level improves some quality metrics of the system without requiring any changes in the software, that is, the hardware is adapted in conformance to the HW-SW interface (e.g., instruction-set architecture). In [Casas et al., 2007] a self-adaptive hardware architecture is presented; this architecture provides self-configuration, self-repair and/or fault tolerance capabilities by means of self-placement and self-routing. In [Bauer et al., 2007] a self-adaptive embedded processor is described. This processor is able to deploy different special instructions at run-time; the decision on which special instructions to deploy and when, is based on their monitored usage. A compile-time analysis of the applications is performed to reduce run-time overhead: the information extracted from this analysis is used to forecast the kind of instructions that will be used by the applications in the immediate future. Thus, self-adaptation can happen without introducing delays in the computation.

2.1.2 Separation of concerns

Separation of concerns between the regular system functionality and the adaptation processes is about putting different concerns into different components that will address them independently; this approach, even though not essential for self-adaptivity, is very important as it offers benefits in terms of generality, level of abstraction, integrated approach, and scalability. In [Kramer and Magee, 2007] a vision of architecture-based self-adaptation is provided and a reference software architecture is proposed.

In [Karsai et al., 2001] another architecture for software self-adaptivity is presented; one of its main goals is separation of concerns. Thus, a *ground-level* that includes baseline processing and a *supervisory-level* that is responsible for adaptation and reconfiguration are considered. The former provides components that are highly optimized for specific situations; the latter select the optimal components for the different situations. The adoption of the supervisory-level enables the system to provide flexibility and robustness. [Schantz et al., 2006] implements a quality of service (QoS) management framework in a distributed system where adaptation strategies are separated from the core functionality by means of aspect languages and an encapsulation model for packaging adaptive behaviors. A standardized way to manage self-adaptivity at application level is provided in [David and Ledoux, 2003], which proposes *separation of concerns* between adaptation management and system functionalities. Self-adaptivity is obtained by applying a set of *adaptation policies* on software components, while these policies are triggered by certain configurable system events. Possible adaptations for component behavior and application parameters are also discussed. Unfortunately, the authors do not discuss if and how a general goal is achieved.

2.1.3 Adaptation management

Most approaches proposed in the literature use a centralized controller for self-adaptation. For example, [Neema and Lédeczi, 2001] proposes a centralized controller based on constraint-guided DSE. The proposed approach is to use models to represent the different points in the design space of the application. The design space is composed of different software component alternatives. The basic idea is to create multiple-aspect models of the design points at design time. These models, along with system constraints, are then embedded into the run-time system and used for self-adaptivity decisions. Each constraint can be associated with one or more values that are continuously measured at run-time. Whenever one of these values crosses the threshold associated with it, the controller is triggered

and the constrained DSE starts.

A different approach is to use decentralized controllers instead of a centralized one. This idea is mentioned in [Vaughan and Munro, 2000]; its main goal is to propose a software architecture that enables applications to be self-tuning and persistent. The work relies on strictly defined and controlled layering of policies and mechanisms, and on the complete control of all layers. Layer coordination is also utilized to obtain a stable behavior of the software. In [Derin et al., 2009], we adopted a similar approach for designing adaptation controllers for multiple goals. We showed (by means of a high level simulation of a self-adaptive system in SystemC) that two independent controllers that are assigned different types of goals may lead to non-converging adaptations. The system is prone to such situations especially when the control decisions affect the monitored parameters of other controllers. We proposed a recommender module that coordinates the two controllers such that both goals are achieved. In our solution, the controller at the higher level recommends to the lower level controller to take a non-greedy decision. [Schantz et al., 2006] uses a mix of centralized and localized QoS management in a distributed real-time system setting. Central control drives the QoS management via policies throughout the network whereas local control is guided by the contract attached to the network component.

2.1.4 Adaptation requirements specification

Adaptation requirements have been specified differently in various early work. In [Neema and Lédeczi, 2001] they are specified as constraints by Object Constraint Language (OCL); in [Schantz et al., 2006] they are expressed as policies via rule-based contracts. In [Hawthorne and Perry, 2004] adaptation requirements are defined as constraints in a custom requirement description language (RDL). In [Brown et al., 2006] the authors introduce a method to specify adaptation requirements by means of goals. Goals are represented by using a graphical language named KAOS; by using this language a full goal-oriented specification of an adaptive system can be drawn.

2.2 MPSoC programming models

An MPSoC-based system is fundamentally composed of a hardware architecture consisting of a set of processing, storage, communication elements which are put together on the same chip, and of the software running on such architecture so as to carry out a function at the same time satisfying some stringent design goals

that would be otherwise not possible with conventional computing platforms. An MPSoC design methodology can be defined as a design flow and its associated design tools that address the creation of MPSoC-based systems. However the design methods are shaped not only by functional constraints but also by the non-recurring engineering (NRE) and manufacturing costs as well as time-to-market. Electronic system-level design (ESL) thrives to address these constraints by automating as much as possible the design process from the functional specification to the final HW/SW based system as a series of refinement steps. Transaction level modeling (TLM) based on SystemC [IEEE Standards Association, 2012] is used widely in the industry to model the abstraction levels above register-transfer level. Several system-level design approaches such as platform-based design, component-based design, design space exploration frameworks emerged to fulfill the promise of ESL [Kogel et al., 2006]. Component-based design is a bottom-up approach that synthesizes application-specific MPSoCs from a library of parameterized intellectual property (IP) cores. Design space exploration is an iterative top-down approach that explores for a given application model various compositions of architectural elements and the mapping of the application onto them. Platform-based design is a meet-in-the-middle approach which identifies common hardware and software features that can be reused in many products within a product line or product family, and aggregating them into a platform [Bailey et al., 2005]. All of these approaches favor reuse of hardware and software to various degrees.

The most prominent feature of an MPSoC is that it is a parallel computing architecture. Parallelism, though, is an aspect that crosscuts all the layers from the application down to the hardware components. The application model helps to exploit the parallelism across these layers during the refinement of the MPSoC-based system. In most system-level design approaches, application specification is done in adherence to a model of computation (MoC). An MoC is the set of rules that defines the formal semantics of interactions and sychronization of concurrent processes in a system [Jantsch, 2003]. The choice of MoC has implications on the rest of the design flow. Therefore most design methodologies are MoC-specific, requiring the application to be specified with a particular MoC. There have been also some efforts such as the Tagged Signal Model [Lee and Sangiovanni-Vincentelli, 1998] and the Rugby metamodel [Jantsch, 2003] that aim at providing a unified theory of MoCs and using them in a hybrid manner. Some examples of common MoCs used in design environments are StateCharts, Petri nets, hierarchical communicating finite state machines, synchronous data flow and KPN [Marwedel, 2011]. Application developers choose the MoC that best fits their application domain. Due to their suitability for embedded sig-

nal processing and multimedia applications, KPN has been adopted widely as a programming model in MPSoC design frameworks such as DOL [Thiele et al., 2007], MAPS [Ceng et al., 2008] and Daedalus [Nikolov et al., 2008]. These design environments incorporate also tools to convert an application specified as a sequential C program into a KPN-based specification.

Considering the case of programming a fixed platform, in addition to these MoC-based approaches, there have been some other approaches that extend capabilities of sequential languages with features that exploit the available parallel computation power in multi-core platforms as well as heterogeneous platforms consisting of CPUs and GPUs. Some of these extensions aim at providing a fixed API that hide the intrinsic platform details from the programmer, thus enhancing code reusability such as OpenMP (an API based on shared memory model) [Dagum and Menon, 1998], POSIX threads (a multi-threading API based on shared memory model) [Nichols et al., 1996], MPI (an API for message passing in high performance computing clusters) [Pacheco, 1996], MCAPI (a API standardization effort for multi-core communications) [The Multicore Association, 2011] and CUDA (an API for Nvidia GPUs exploiting data and task parallelism) [Nvidia, 2014]. In addition, OpenCL (Open Computing Language) is an open royalty-free standard that consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language that supports both data-/task-based parallel programming models [Howes and Munshi, 2014].

The underlying MPSoC architecture and its memory organization (e.g., shared vs. distributed, NUMA vs. NORMA) plays an important role when choosing the programming model. The multi-core APIs such POSIX, MPI and MCAPI deal with concurrency at a low abstraction level. Therefore using them is more tedious than adopting an MoC by which parallel programs can be generated correctly by construction. However these APIs can be used to implement MoC semantics on supported platforms. In fact, the KPN middleware described in section 3.2.2 is built on top of an MPI-like interface. The strength of the GPU programming models such as CUDA and OpenCL lie in their ability to exploit fine-grained data parallelism. In the case of a GPU-based platform, a KPN process can leverage this ability by being implemented in CUDA or OpenCL. In fact such efforts aiming at generating data parallel programs for GPUs from KPN models are present Balevic and Kienhuis [2011].

The selection of MPSoC architectural template, memory organization and programming model has a deep impact on the type of problems to be addressed during the design flow and on the way the mechanisms for system adaptivity and fault tolerance may be implemented. For example, in the case of the NUMA

model, data distribution and affinity-based scheduling problems are relevant [Nikolopoulos et al., 2001]; whereas in the NORMA model, the corresponding problems are communication minimization and task mapping. As another example, when implementing task migration in the NUMA model, a task's state is reachable from the destination node; whereas in the NORMA model, it has to be explicitly transferred to the destination node. The reference architecture used in this thesis, which is based on the NORMA model, is described in chapter 3.

In this thesis, we adopt the PPN MoC, which is a subset of KPN. The KPN and PPN models will be described more formally in the next two sections based mainly on [Jantsch, 2003] and [Verdoolaege, 2010], respectively.

2.3 Kahn Process Networks

When creating distributed programs, nondeterminism poses a great challenge due to the different delays that may occur in computation and communication when the program is run on the final platform. Being an untimed MoC based on asynchronous message passing, Kahn Process Networks guarantees functional correctness independent of timing. A KPN is an arbitrary composition of concurrent processes connected through channels: each channel may have a single writing and a single reading process. The basic piece of data exchanged via channels is called a token. Each channel carries data of a specific type (e.g., integer, boolean, complex data types etc.). The channels are unbounded FIFOs which can be written in a non-blocking manner. However the reading process blocks when there is no token in the channel's FIFO. Figure 2.1 shows a KPN process example with one input and one output channel. Defining the history of a channel as the stream of tokens that passes through it, a process function is said to be determinate if the histories of its output channels depend only on the histories of its input channels. It is important to note that the process function is not a mapping from input token values to output token values, rather from histories of token values to histories of token values. Therefore processes can be stateful. A KPN process should also satisfy the continuity and monotonicity properties.

More formally, let D be the set of all possible token values and D^w be the set of all finite and denumerably infinite streams over D, a KPN process with k input and l output channels is a mapping $P : (D^w)^k \rightarrow (D^w)^l$.

Defining the prefix ordering relation between two streams $S, S' \in D^w$ by

$$S \sqsubseteq S' \Leftrightarrow S \textit{ is an initial segment of } S',$$

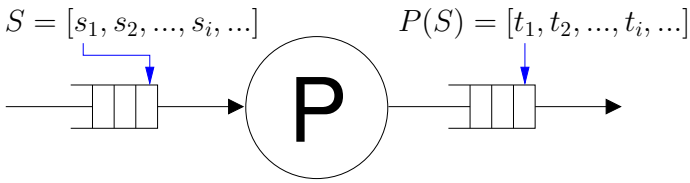


Figure 2.1. A KPN process with single input and output channels

a process P is monotonic if

$$S \sqsubseteq S' \Rightarrow P(S) \sqsubseteq P(S').$$

Monotonicity implies that a process can start producing output without waiting for all the inputs to be available. The new input will only add new output to what has already been processed.

The prefix ordering relation is a partial order on D^w with empty stream ([]) as the minimal element. Any increasing chain $C = \{C_i : C_i \sqsubseteq C_{i+1}, \forall i > 0, C_i \in D^w\}$ has a *least upper bound*, represented by $lim(C)$. This makes D^w a *complete partial order*.

A monotonic process P is continuous if

$$P(lim(C)) = lim(\{P(C_i) : C_i \in C\})$$

Continuity implies that, when processing infinite input streams, a process must be able to process finite substreams without requiring to read infinite streams in order to produce output. Continuity is a stronger restriction on the process than monotonicity. A continuous process is always monotonic.

These properties can be generalized for KPN processes with multiple input and output channels by extending the definition of the prefix ordering relation for a pair of stream lists, $\mathbf{S} = [S_i : 1 \leq i \leq k]$ and $\mathbf{S}' = [S'_i : 1 \leq i \leq k]$ as the following

$$\mathbf{S} \sqsubseteq \mathbf{S}' \Leftrightarrow S_i \sqsubseteq S'_i \qquad \forall i, 1 \leq i \leq k.$$

KPN processes are closed under arbitrary composition. Meaning that, a network of such processes results in the final KPN to be determinate, monotonic and continuous as well.

KPN model has several advantages. The scheduling of processes does not effect the functional behavior, in fact, KPN can be self-scheduled in a data-driven manner. It is suitable for stream processing and allows exploiting task-level parallelism, thus making it favorable for mapping onto distributed multi-processor platforms. However the unbounded FIFO requirement hinders its applicability

in system design since it would require, in the worst case, a memory of infinite size. Therefore several variations of KPN have been proposed (e.g., Synchronous Data Flow [Lee and Messerschmitt, 1987b] and Polyhedral Process Networks [Verdoolaege et al., 2007]) in order to increase the analysis and synthesis capabilities when using the KPN model.

2.4 Polyhedral Process Networks

Polyhedral Process Networks is a variant of KPN in which buffer sizes of the channels can be determined at design time, unlike the case of general KPN programs. The `pn-compiler` [Verdoolaege et al., 2007], which generates a PPN from a given sequential code that is composed of static affine nested loops, can derive for each channel a buffer size which guarantees deadlock-free execution of the PPN. For a theoretical background on the derivation of a PPN model from static affine nested loop programs,

Similar to KPN, a PPN is a graph defined as a tuple $(\mathscr{P}, \mathscr{C})$, where:

- $\mathscr{P} = \{P_1, \ldots, P_N\}$ is a set of processes;
- $\mathscr{C} = \{ch_1, \ldots, ch_K\}$ is a list of FIFO channels.

Each process $P \in \mathscr{P}$ has a set of input channels IC_P and output channels OC_P. PPN processes communicate and synchronize using these FIFO channels. The PPN semantics forces a process to *block on read*, when trying to get a token from an empty FIFO, and *block on write*, when trying to write a token to a full FIFO. The processes which write into IC_P are the *predecessors*, the processes which read from OC_P are the *successors*. The producer process, which writes data to a channel ch, and the consumer process, which reads data from it, are denoted respectively as $prod(ch)$ and $cons(ch)$.

All PPN processes have the same code structure, an example of which is given in figure 2.2(b). Nested loops iterate, for a given number of times, the body of the process, which is split in 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 figure. 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 (*write* statement). The read and write statements can be guarded by conditions that can be defined only in terms of the iterator values (e.g., i, j) and loop parameters (e.g., M, N). Therefore, control flow of PPN processes is static, meaning that the token consumption and production

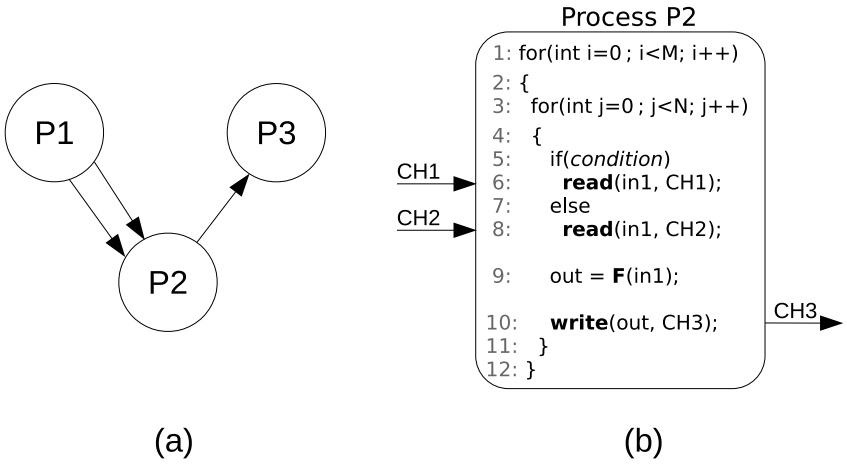


Figure 2.2. Example of a PPN (a) and structure of process $P2$ (b).

rates should not depend on the input data stream. All loop bounds, conditions, array index expressions, if any, should be representable as affine constraints. The process function should be pure, meaning that values of iterators or arrays cannot be changed. Given that these requirements are met, such processes can be analyzed by means of polyhedra. A polyhedron is a set of rational values described by affine constraints. That is, $\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{Ax} \geq \mathbf{b}$ represents a polyhedron in n-dimensional space where A is a $m \times n$ matrix and b is a m-dimensional vector of integers. The iterator domain of a process is the set of all possible values that its iteration vector can have. For a process $P_i \in \mathscr{P}$ that has d nested loops (i.e., d iterators), the iterator domain can be represented as a polyhedron $D_i = \{\mathbf{x} \in \mathbb{Z}^d \mid \mathbf{Ax} \geq \mathbf{c}\}$. For example, the iterator domain of the process P_2 in figure 2.2(b) is defined by $D_2 = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i < M \wedge 0 \leq j < N\}$ where M and N are constant integers. The conditions guarding the read and write statements on a channel add additional constraints to the iterator domain and define the target and source domain of the channel, respectively. Such a formalization of the PPN processes allows design time analysis (e.g., buffer size computation) [Verdoolaege et al., 2007] and HW/SW co-design of PPN applications [Nikolov et al., 2008].

2.5 KPN for MPSoCs

Previous research on the use of KPN for multiprocessor embedded devices mainly focused on the design of frameworks which employ them to model the applica-

tion [Stefanov et al., 2004; Nieuwland et al., 2002; Kwon et al., 2008], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [Bacivarov et al., 2010; Haid et al., 2009]. In [Stefanov et al., 2004; Nieuwland et al., 2002], different methods and tools are proposed for automatically generating KPN application models from programs written in MatLab or C/C++. More specifically, the `pn-compiler` tool [Verdoolaege et al., 2007] is used to automatically convert static affine nested loop programs to parallel PPN [Verdoolaege, 2010] specifications and to determine the buffer sizes that guarantee deadlock-free execution. Design space exploration tools and performance analysis are then usually employed for optimizing the mapping of the generated KPN processes on a reference platform. A design phase usually follows in which software synthesis for multi-processor systems [Kwon et al., 2008; Haid et al., 2009], or architecture synthesis for FPGA platforms [Stefanov et al., 2004] is implemented. A survey of design flows based on the KPN MoC can be found in [Haid et al., 2009].

The trend from single core design to many core design has forced to consider inter-processor communication issues for passing the data between the cores. One approach is synthesizing the software relying on the high level APIs provided by the reference platform for facilitating the programming of a multiprocessor system. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [The Multicore Association, 2011] that targets the inter-core communication in a multicore chip. MCAPI is the lightweight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [*A High Performance Message Passing Library*, n.d.]. However these MPI standards are not quite fit for the KPN semantics and building the semantics on top of their primitives is less efficient compared to platforms with dedicated FIFO support.

The communication and synchronization problem when implementing KPNs over multi-processor platforms without hardware support for FIFO buffers has been considered in [Nadezhkin et al., 2009] and [Haid et al., 2009]. In [Nadezhkin et al., 2009] the *receiver-initiated* method has been proposed and evaluated for the Cell BE platform. On the same hardware platform, [Haid et al., 2009] proposes a different protocol, which makes use of mailboxes and *windowed FIFOs*.

In [Nejad et al., 2009] the problem of implementing the KPN semantics on a NoC is addressed. However, in their approach the NoC topology is customized to the needs of the application at design time and network end-to-end flow control is used to implement the blocking write feature. Therefore it cannot support the online remapping of tasks.

An approach to guarantee blocking write behavior is also used in [Almeida

et al., 2009]. That work uses dedicated operating system communication primitives, which guarantee that the remote FIFO buffer is not full before sending messages through a simple request/acknowledge protocol.

2.6 Mapping applications to NoCs

With the emergence of NoCs the problem of optimally mapping tasks on top of NoC architectures has been the subject of a significant amount of research [Singh et al., 2013]. While the optimal distribution of tasks on a given MPSoC architecture at design time (*static mapping*) is a research problem that has been addressed and solved by several authors [Lei and Kumar, 2003; Thiele et al., 2007; Amory et al., 2011], the optimal online task mapping (*dynamic mapping*) presents a significant amount of challenges that still need to be fully addressed by researchers [Walter et al., 2009; Chou and Marculescu, 2011]. This is particularly true in the case of the onset of runtime permanent faults in the processing elements, when a redistribution of the tasks executing on the faulty cores is needed in order to provide a graceful degradation of the performance of the platform.

In most of the related work (e.g., [Lei and Kumar, 2003; Jena and Mahanti, 2008]), the problem of static task mapping has been addressed in two phases. The first phase addresses the partitioning problem, taking as input the task graph and a list of IPs and providing as output the core communication graph (CCG). A vertex in the CCG represents a core where one or more tasks of the given task graph are merged into one with a schedule. This step is usually performed by exploring the design space through genetic algorithms for selecting the mapping that, given some assumption of the average system delay, optimizes the computation [Lei and Kumar, 2003]. The second phase is the core mapping. It tries to optimally bind the vertices of the CCG to specific nodes of the NoC, selected amongst the group of IPs that can execute the specific task. Core mapping problem starts with the CCG and results in a mapping of cores to tiles. Goal of this second phase is usually to minimize the communication cost.

ILP-based solutions to similar problems have been proposed in the case of contention-aware application mapping on NoCs [Chou and Marculescu, 2008]; floor-planning and topology generation for NoCs [Srinivasan et al., 2006]; task mapping and scheduling on multi-core architectures [Yi et al., 2009]; task mapping on shared memory bus-based heterogeneous MPSoCs [Erbas et al., 2006]. It is possible to classify the previous work in terms of the tackled problem (core mapping, task mapping, partitioning, allocation, scheduling, routing, topology generation); optimization goals (execution time, delay, communication, power,

robustness, contention, flexibility); optimization techniques (heuristics, evolutionary algorithms, exact solutions); architectural platform (fixed/free NoC topology, fixed/free routing).

The optimal mapping of application tasks on MPSoC platforms is also targeted in [Thiele et al., 2007] and [Le Beux et al., 2010]. Both proposed approaches apply an optimization based on the use of genetic algorithms for automatically exploring the design space; the former optimizes the total execution time and communication load; the latter optimizes the throughput, area and flexibility.

Other than the optimal solutions, several work have proposed as possible solution to the optimization problem the use of heuristics [Hu and Marculescu, 2003; Murali and De Micheli, 2004; Hu and Marculescu, 2005; Marcon et al., 2005; Wang et al., 2010], evolutionary algorithms [Ascia et al., 2004; Zhou et al., 2006; Jena and Mahanti, 2008; Bhardwaj and Jena, 2009; Walter et al., 2009; Fekr et al., 2010] and a mix of both [Srinivasan and Chatha, 2005; Modarressi and Sarbazi-Azad, 2007]. Heuristics are in fact needed when the dimension of the NoC increases. The ILP formulation is too time-consuming to be solved with current ILP solver software tools, in particular when considering its possible use at runtime for calculating the remapping of the tasks running on a processor that becomes faulty. The usual approach followed in the mentioned related work is to compare the performance of the proposed solutions with each other, or with a solution found by applying simulated annealing [Ababei and Katti, 2009].

2.7 Task migration

In this thesis, we view fault tolerance as an aspect of run-time management and aim at achieving it by remappping of tasks using task migration. Process migration mechanisms [Smith, 1988; Milojičić et al., 2000] have been widely studied and implemented in the context of distributed computing systems to enable dynamic load distribution, fault resilience, improved system administration and data access locality. In recent years, run-time management of multiprocessor systems has gained popularity also in the embedded systems research area. This domain imposes tight constraints such as cost, power, and predictability that should be carefully taken into account by run-time management mechanisms. In [Nollet et al., 2010], a survey of run-time management applications in state-of-the-art academic and industrial MPSoC solutions is presented together with a generic description of run-time manager features and implementation alternatives. One of the highlighted research challenges, which is the concern of this thesis, is run-time system adaptation for fault tolerance.

Task migration is a specific component of run-time management strategies. Several previous work address the implementation of task (or process) migration in MPSoCs. Task migration approaches are explained and quantitatively evaluated in [Bertozzi et al., 2006] and [Acquaviva et al., 2008]. Dynamic task remapping is achieved at user-level or middleware/OS level respectively. In both of these approaches, the user needs to define checkpoints in the code where the migration can take place. This can require some manual effort from the designer. Moreover the inter-task communication realization exploits a shared memory system.

The closest to our work is [Almeida et al., 2009], in which the goals of scalability and load balancing are achieved through a distributed task migration decision policy over a purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network MoC. In their approach the actual task migration can take place only at fixed points, which correspond to the communication primitive calls. This method assumes that the computation relies on a strict consumer/producer model where no internal state is kept from iteration to iteration. This translates in the fact that there cannot be any dependencies between two adjacent computed data chunks. We adopt a similar application model (see section 3.2.1) in the recovery technique proposed in section 5.3 and adaptation techniques proposed in chapter 6.

One of the implementation decisions for task migration is the migration of task's code. In the code replication approach [Pittau et al., 2007], each node stores a copy of tasks' code. Whereas in the code recreation approach [Almeida et al., 2009], code of the tasks is also migrated alongside their state. The former approach requires additional memory but it reduces the migration time.

2.8 Fault tolerance

In this section, we strive to provide some background on hardware fault tolerance followed by the related work. For more details on fault tolerance, readers can refer to the excellent books by Koren and Krishna [2007]; Mukherjee [2008]; Sorin [2009].

Fault tolerance consists of two steps: detection and recovery. Detection identifies the presence of a fault or an error. Recovery transforms a system state affected by one or more errors (due to the underlying presence of faults) into a state without detected errors (error recovery) and such that previously detected faults will not be activated again (fault handling) [Avizienis et al., 2004]. In essence, tolerance to hardware faults is achieved by means of redundancy either

in time, hardware or information regardless of the abstraction level at which it is being used.

A vast amount of research effort has been spent on hardware fault tolerance which addressed it at different levels of abstraction. A survey of techniques up to the processor level can be found in [Koren and Krishna, 2007]. Gizopoulos et al. [2011] presents a survey of fault tolerant architectures in multi-core systems. In the following, firstly in section 2.8.1 and 2.8.2, we provide some background on fault detection and error recovery techniques, respectively. In section 2.8.3, we survey related work on system-level fault tolerance in embedded systems. Lastly, in section 2.8.4, we look at the related work with respect to lifetime reliability.

2.8.1 Fault detection

There are mainly four detection approaches for processor faults, namely, redundant execution, dynamic verification, built-in self-test (BIST) and anomaly detection [Gizopoulos et al., 2011]. They vary in terms of their hardware cost, performance overhead, detection latency, targeted faults and fault coverage. Redundant execution can be used as an architectural solution (e.g., lockstep cycle-by-cycle checking, redundant multi-threading in a single core or a multi-core setting) or as a software-based solution (e.g., duplicated instructions). Dynamic verification employs concurrent checking of specific invariants such as control flow, data flow and computation via dedicated hardware checkers [Meixner et al., 2007; Ananthanarayan et al., 2013]. When targeting permanent faults, the use of concurrent detection techniques, which target permanent as well as transient faults, is an overkill. However it comes with the benefit of detecting faults with a small detection latency. BIST and anomaly detection techniques impose a longer detection latency which creates a window for error propagation. Software anomaly detection monitors symptoms of faults in the form of anomalous software behavior such as hardware fatal traps, kernel panics and application aborts [Li et al., 2008]. BIST is a design-for-testability practice used for manufacturing testing. Test patterns are applied to the device under test by dedicated testing hardware units. Software-based self-testing is a low cost alternative as it removes the need for test pattern generation and storage by means of a self-testing routine. Software-based self-testing has been used recently in an online and periodical manner to detect operational permanent faults [Constantinides et al., 2007].

2.8.2 Error recovery

Error recovery techniques can be classified into two categories: forward error recovery (FER) and backward error recovery (BER). In FER, the system continues to make forward progress by masking the fault and correcting the error without any re-execution of data preceding the fault [Pradhan and Vaidya, 1994; Xu and Randell, 1996]. This is mainly achieved by spatial redundancy techniques such as N-modular redundancy (NMR) or fail-over. Although implementation of these techniques is rather easy, the overhead associated with them are substantial. On the other hand, BER makes use of redundancy in time by returning to a state saved in a stable storage before the occurrence of the fault and continuing operation by re-executing from that point (checkpoint and rollback) [Elnozahy et al., 2002]. Although its implementation is more tedious in general, it helps avoiding higher overheads of the FER techniques. BER has been studied widely in shared memory systems [Sorin, 2009] as well as message-passing systems [Elnozahy et al., 2002].

In the case of shared memory multiprocessor systems, checkpointing-and-rollback recovery has been implemented in software or hardware. Software-based solutions take checkpoints at the application level [Bronevetsky et al., 2004] or in an application-indifferent manner (known as system level checkpointing) [Litzkow et al., 1997; Dieter and Lumpp Jr, 1999; Hargrove and Duell, 2006]. Hardware-based solutions take a global system-wide checkpoint stored in the main memory [Prvulovic et al., 2002] or local, coordinated checkpoints stored in special log buffers [Sorin et al., 2002]. These solutions have been coupled with various fault detection mechanisms such as online software-based self-testing in [Constantinides et al., 2007] and software-based anomaly detection in [Sastry Hari et al., 2009]. The performance overhead as well as the additional memory required for checkpoints depend on the checkpointing interval and the application.

In the case of message-passing systems, BER has been implemented in software and mostly for long running scientific applications in high performance computing [Schulz et al., 2004; Bouteiller et al., 2006]. BER techniques have a few variations. In uncoordinated checkpointing, processes decide to take a checkpoint by themselves. However, when a fault occurs, finding the set of consistent checkpoints that reflects a snapshot of the entire system (called a recovery line) requires an online algorithm which would increase the recovery time. Moreover, it is prone to the domino-effect which may lead to rolling back all the way to the beginning of the computation. Most importantly, each process has to keep several checkpoints, which would result in a memory overhead. Coor-

dinated checkpointing, on the other hand, relies on storing a single consistent checkpoint by means of a coordination protocol where a coordinator initiates a global checkpoint by synchronizing other processes. Again, this coordination mechanism leads to an overhead in the steady-state operation of the system. Alternatively, in log-based BER, each process can take a checkpoint without any coordination but they store the history of all incoming messages after a checkpoint. Upon fault detection, an online algorithm is used by each process to find the checkpoint to rollback to by coordinating with its predecessors. Differently than normal operation, instead of resuming execution by exchanging messages, the processes play the logged messages to create the exact state at the time of the fault without having to synchronize on the messages from other processes. Therefore a faster recovery can be achieved at the expense of the memory overhead required to log the messages. When adopting checkpoint-based rollback recovery systems, several implementation issues arise such as what to checkpoint, how to take a checkpoint for a given processing core and where to checkpoint. The final overhead is determined by aggregating the effects of the answers to these questions. Checkpointing support at the processor, compiler or kernel level can result in a more efficient and easy implementation. However, the size of the checkpoint increases at lower abstraction levels making application-level checkpointing more favorable [Schulz et al., 2004].

2.8.3 Related fault tolerance approaches in embedded systems

Although BER and FER solutions explained in the previous section are valid for message passing systems in general, when applied to more constrained systems such as embedded multicore platforms, the resource overhead has to be considered more carefully because they may impose quite large overheads both in performance and power due to checkpointing or replication.

In the following, the related work on fault tolerance that target embedded systems are presented, in particular, focusing on system-level approaches and their recovery techniques.

Task-level active spatial redundancy

Fault tolerant mapping of applications on MPSoCs has attracted some attention recently [Saraswat et al., 2010; Huang et al., 2011; Stralen and Pimentel, 2012; Bolchini and Miele, 2013; Kang et al., 2014a,b]. Most of these approaches focus on transient errors and consider real-time applications, hence they address both mapping and scheduling problems.

Bolchini and Miele [2013] propose a reliability-driven system-level design methodology for mixed-critical embedded systems against transient faults. The approach aims at implementing a performance-optimized hardened implementation of the system by means of a DSE process that selects the appropriate technique (or set of techniques), possibly exploiting also the fault management features provided by the target architecture. Three criticality levels that can be assigned to a task are fault-tolerance, fault-detection and fault-ignorant. A task is hardened according to its criticality requirement either by a task-level redundancy technique such as task duplication and NMR, or by mapping it on a HW resource hardened by fault-tolerant fabric, concurrent error detection or checkpoint-and-rollback capabilities. The genetic algorithm based DSE evaluates design points only in terms of execution time metric by associating some penalty delays to actions such as voting and rollback.

In [Kang et al., 2014a], a design space exploration for the mapping of mixed critical applications on MPSoCs is proposed which achieves fault tolerance by three techniques: task level NMR (refered to as active redundancy), duplex with spare (refered to as passive redundancy), and re-execution. Re-execution assumes that an error is detected at the end of each execution of a task. Since transient errors are instantaneous, this requires some sort of concurrent error detection. However no reference is made to a particular error detection technique. Tasks are specified by a reliability constraint (in terms of failures in time). The DSE results in hardened applications optimizing power, reliability and real-time behavior. In [Kang et al., 2014b], same authors add the capability of dropping non-critical tasks in order to keep satisfying the deadlines of higher criticality tasks when errors occur. This capability is added as an extension to the scheduler on each processor and is built on top of the static hardening scheme described in their previous work. The exploration framework relies on analytical models and uses genetic algorithms.

Analysis and optimization of fault-tolerant task scheduling for multiprocessor embedded systems is also addressed by [Huang et al., 2011]. System-level reliability in the presence of software/hardware redundancy is computed through the implementation of a Binary Tree Analysis based on a set of existing fault- and process-models. The Binary Tree Analysis is integrated into a multi-objective evolutionary algorithm which performs the reliability-aware design optimization, providing as result the mapping of tasks to processing elements, the exact task and message schedule, and the fault-tolerance policy assignment. Different than the abovementioned approaches, this approach addresses permanent faults by integrating static schedules computed at compile time for all permanent fault scenarios. The number of these schedules increases dramatically with problem

size similar to [Lee et al., 2010].

Pinello et al. [2008] demonstrate a design flow that achieves fault tolerant distributed deployment of safety critical embedded software for automotive applications. They address permanent and transient faults. In doing so, they propose a new fault tolerant data flow model that expresses redundancy through replication of one or more tasks.

Task re-execution

Apart from the exploration frameworks mentioned earlier [Bolchini and Miele, 2013; Kang et al., 2014a] which also support re-execution, Izosimov et al. [2012] use re-execution as the recovery technique for scheduling and optimizing fault tolerant systems with transparency/performance trade-offs. They focus on multiple transient errors. The recovery overhead includes the time that is needed in order to restore task inputs, clean up the processing node's memory, and restart task execution. Transparency of faults between dependent tasks is achieved by allowing a long enough time to recover from errors such that the schedule of consumer tasks is not affected. The approach assumes the availability of an error detection technique that detects a transient error at each execution of a task. Similarly, Mossé et al. [2003] propose to leave sufficiently long gaps between tasks so that faulty tasks, which experience transient faults, can be rescheduled for re-execution without violating deadline guarantees.

Checkpointing and rollback

In a similar context to ours, Stralen and Pimentel [2012] aim at achieving tolerance to transient errors when running KPN applications in MPSoCs. They present a framework that explores fault-tolerant mappings using genetic algorithm based on a simulation model. It evaluates the impact of task redundancy (with TMR and DMR) on the number of dropped frames, which represents a measure of performance and reliability combined, and power figures of a given application. TMR achieves fault detection and correction at the same time while, in the case of DMR, an error is detected by the mismatch in DMR's checker, and error recovery is done by a checkpoint-and-restart method that is used within the DMR-ed subnetwork. This approach requires for each redundant task the presence of a message cache that is able to store all the messages in between checkpoints.

Drop and forward recovery

Error tolerant computing aims at exploiting the capability of an algorithm to tolerate errors that occur during computation [Li and Yeung, 2007]. When running error-tolerant applications, some errors can be simply allowed to propagate. In [Lee et al., 2008], authors propose a cross layer approach in order to mitigate transient errors occuring in data caches when running multimedia applications. In particular, they explore a Drop and Forward Recovery mechanism that drops a current encoding frame and moves forward to the next frame once an error is detected in a mobile video encoding system. Using error detection codes at the hardware layer combined with failure handling capabilities implemented at the middleware and application layers allows achieving better performance, energy and reliability trade offs compared to the use of error correcting codes in the data cache, which is a pure hardware approach. In [de Kruijf et al., 2010], authors extend the instruction-set architecture of a processor and the programming language to support recovery blocks which are executed if a soft error is detected during the execution of the preceding code block encapsulated by a relax statement. Beside supporting a retry statement in the recovery block that re-executes the relax block, they also enable roll-forward recovery by allowing an erroneous value to propagate as the result of the code block or by escalating the error to the function caller. This is possible only in a functional programming style in which the relax block does not have side effects.

Core remapping

Core remapping (or core sparing) has emerged firstly as an architecture-level defect tolerance technique for the purpose of increasing the yield after manufacturing testing of the chip by replacing defective cores with spares [Greene and El Gamal, 1984; Collet et al., 2011; Zhang et al., 2008]. This approach has later been extended as a run-time management technique in order to address permanent faults occuring in operational conditions [Ababei and Katti, 2009; Chou and Marculescu, 2011]. These core remapping approaches remap a core (i.e., a scheduled set of tasks mapped on the same processing node, as described in section 2.6) as a whole to a spare processing element. Considering the related work specifically addressing strategies for increasing fault tolerance in case of faulty cores in NoCs, in [Ababei and Katti, 2009], authors investigate the use of adaptive remapping for moving tasks running on a core found faulty on a different spare processing element. Their approach aims at minimizing the communication volume after remapping.

In [Chou and Marculescu, 2011], 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, 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.

Although core remapping is a way of tolerating permanent faults, it is not a complete fault tolerance solution by itself. It has to be accompanied with proper fault detection and recovery techniques. The abovementioned approaches fall short of addressing these problems.

Core salvaging

Apart from the system-level techniques presented above, we find it worthmentioning that a few hardware-based reconfiguration techniques have exploited inherent intra-core and cross-core redundancy at micro-architectural and architectural levels. From microarchitectural standpoint, some approaches make use of the redundant resources available inside a core by disabling defective execution pipelines [Schuchman and Vijaykumar, 2005] or scheduling operations on fault-free intra-core resources [Meixner and Sorin, 2008; Shivakumar et al., 2003; Srinivasan et al., 2005; Bower et al., 2004]. Especially in the case of complex superscalar architectures, which contain non-essential components (e.g., branch prediction unit), it is possible to avoid using a defective unit or carry out its function via some other units. The core is still able to implement its instruction set architecture, though with degraded performance. These techniques introduce modifications to the microarchitecture of the core, hence increasing its complexity.

Architectural core salvaging techniques exploit cross-core redundancy. Core cannibalization [Romanescu and Sorin, 2008] and StageNet [Gupta et al., 2008] approaches allow shared use of some pipeline stages between faulty and fault-free cores. In [Powell et al., 2009], the threads are relocated to another core when encountering instructions that use a defective unit in the core. The remapping policy aims at finding a mapping that minimizes the number of migrations between the defective core and fault-free cores. Authors also propose policies that take into account the type of defective units. For example, considering a defective branch predictor, the remapping policy assigns to the defective core the threads for which branch prediction performs badly. Although core salvaging techniques allow using defective cores, they work if the defects appear only in

specific units of the core, hence resulting in a small fault coverage. Therefore they require additional techniques to implement full coverage.

Task remapping

The approach adopted in this thesis is task remapping. It is a graceful perfomance degradation technique that relies on using as much as possible the slack time available in the computational resources to compensate for the faulty processing element by redistributing its tasks to fault-free nodes. Unlike core remapping, it does not require spare cores. It is based on core disabling (i.e., turning off and isolating defective cores). The system continues operation with a lower number of cores.

Scheduling is a widely researched topic in general purpose distributed computing systems [Casavant et al., 1988]. In that context, we encounter the earliest examples of a similar problem in [Chou and Abraham, 1983; Patnaik and Iyer, 1986; Singh et al., 1991] where loads are re-scheduled in presence of faulty computing nodes. Fault tolerant scheduling in homogeneous real-time systems is surveyed in [Krishna, 2014]. These approaches are not sufficiently generic to be applicable in our case. Their system models are fundamentally different in several aspects such that tasks do not have precedence relations, computing nodes are homogeneous or only load balancing is taken into account.

[Lee et al., 2010] proposes a task remapping technique for multi-core embedded systems aiming at minimizing the throughput degradation, based on an intensive compile-time analysis for all possible failure scenarios. Pre-computed remapping information is stored and retrieved at run-time for remapping the tasks following the decision taken at compile-time. However, in the case of a restricted amount of local memory in the NoC tiles, or for complex fault scenarios in NoCs of a significant size, the amount of memory needed for storing the remapping information may make this technique not applicable.

Task mapping in the case of defective tiles has been investigated in [Amory et al., 2011], where the generation of a task mapping is evaluated in terms of energy consumption and execution time. This is not a real task remapping approach because it aims at finding a static mapping for a NoC platform which is known to contain defective tiles.

In the particular case of process networks, a fault tolerant process network model is proposed in [Ceponis et al., 2008]. A fail-stop error model is assumed, that is, a process stops execution after experiencing a fault. Failed processes are detected by their predecessor and successor processes via time-outs when blocked on empty or full FIFO channels. Recovery is achieved by merging the

function of the failed process into one of its successor processes. The approach does not clarify some important issues, such as how the data lost during failure are compensated using default values, how the error is contained and how it is guaranteed that the system would not deadlock due to missing data. Since the approach is realized only as a multi-threaded simulation of a simple application, the authors do not provide an evaluation of its performance.

2.8.4 The lifetime reliability aspect

The online task remapping technique enables the system to survive with possibly some performance degradation. This capability of the system increases the number of faults that should occur in order to make it fail, thus increasing its lifetime reliability. In this section, the previous studies that treat lifetime reliability as an optimization goal during task mapping are surveyed.

Due to the ever-increasing possibility of incurring in run-time permanent faults in processors and MPSoC components - faults caused for instance by wearout effect [Borkar, 2005] - system lifetime reliability has been proposed as an explicit metric to be taken into account during the task allocation phase [Huang et al., 2009]. In [Huang et al., 2009], the problem is addressed by generating a unique task schedule with maximum lifetime reliability for a single-mode embedded system. With regard to previous work, [Huang et al., 2009] takes into account aging effects of processors in the calculation of system reliability. In [Huang and Xu, 2010], the same authors extend the methodology proposed in [Huang et al., 2009] by minimizing energy consumption of a MPSoC platform in the case of a given lifetime reliability constraint. In both approaches, task allocation is generated at design stage and a unified task schedule for each execution mode is constructed for all the products. In [Huang, Ye and Xu, 2011], initial schedules generated at design stage are optimized separately with online adjustment at regular intervals for lifetime reliability and/or energy-efficiency improvement.

2.9 Application-level self-adaptation for quality management

Systems are usually expected to operate with a certain quality when delivering their services. Similar to fault tolerance, management of quality of service has benefited from the increased attention to run-time management in MPSoCs. According to [Nollet et al., 2010], which surveys several approaches from academia and industry that address different aspects of run-time adaptation, one of the

two components of a generic self-adaptive run-time environment, other than a resource manager, is the quality manager. In our work, we incorporate quality management support into the NoC-based platform through an application-level self-adaptation framework. Quality management support may reside alongside other self-adaptation services such as the fault-tolerance support described in chapters 4 and 5. In this thesis, we focus particularly on the adaptation of application-level parameters to meet application's performance goals. In the following, we provide an overview of the related work, firstly, on application-level adaptation and secondly, on quality management in multimedia systems.

2.9.1 Adaptation of application-level parameters

An application can be adapted in a number of ways, for example, by changing the value of a variable that appears in a function, or by using a conditional variable to select the concrete implementation of a function. Such parametric adaptations can allow the system to operate at different trade-off points in terms of system's objectives. In order to enable parametric adaptations, the parameters need to be identified and changed consistently throughout the application.

There is a body of work that strives to embed dynamicity into the models of computation in order to increase their expressive power. It is mainly motivated by advanced signal processing applications that require multi-mode, multi-standard, variable data rate or other kinds of adaptable operation. [Bhattacharyya et al., 2013] surveys dataflow models of computation that embrace dynamicity to different extents. These MoCs are namely Boolean dataflow, CAL actor language, parameterized dataflow, enable-invoke dataflow, dynamic polyhedral process networks, scenario aware dataflow, and a stream-based function actor model. Of particular relevance to us is parameterized PPN.

In P^3N [Zhai et al., 2011], a parameterized polyhedral process network model is defined to support run-time parametric adaptations. It uses similar concepts defined in parameterized SDF [Neuendorffer and Lee, 2004] such as quiescent points and reconfiguration ports. Quiescent points are the execution points at which a parameter is reconfigurable. A reconfiguration port is bound to a parameter of its task and tokens received through the port reconfigure the parameter. P^3N enforces through a design time analysis and a run-time checker the consistent changing of multiple parameters by making sure the production and consumption rates of the channels remain equal as a result of the reconfiguration.

By operating at the MoC level, the adaptation of the application can be carried out in an application independent manner. Contrarily, ad hoc solutions may be used to coordinate the reconfiguration in multiple components based on bar-

rier synchronization as examplified in Cholla [Bridges et al., 2009]. In Cholla, barrier members are adaptable components or different portions of adaptable components. The barrier locations in all its members need to be determined according to the application. Moreover, barrier synchronization implies that all the members of the barrier are blocked until they are synchronized and reconfigured.

2.9.2 Quality management in multimedia systems

Self-adaptation has been used in distributed systems for the management of QoS mostly through adaptive middleware mechanisms and custom adaptation management protocols. A control theoretical approach to QoS is proposed in [Li and Nahrstedt, 1998] and [Li and Nahrstedt, 1999]. In the first paper, the authors introduce a passive adaptation task mechanism, located in the middleware level to support application-specific adaptation. Basically, passive adaptations can be viewed as transformations of the data input stream incoming into a task (e.g. a software component) to fit a required QoS. The middleware performs adaptation without a feedback loop between applications and the transport layer based on certain QoS metrics. In the second paper, the same authors extend and improve their technique by means of proportional/integral/derivative (PID) control and fuzzy control models in order to balance and support both application-specific adaptations and system-wide requirements, such as stability and agility of the adaptation and fairness among multiple applications. Cholla [Bridges et al., 2009] presents a Linux-based prototype implementation of a software architecture that controls and coordinates adaptation policy decisions inside network protocols and multimedia applications. Adaptation controller is defined by rule sets each of which implements a portion of the adaptation policy. If rule sets conflict when determining an output variable, additional coordination rules are used for resolution. Rule sets, which are predicate-action pairs, support fuzzy control techniques.

In [Al-Ali et al., 2004] an approach for managing quality of service in grid systems is presented: system resources are managed in an adaptive way both to satisfy the quality of service requirements and to use the resources efficiently. Thus, self-adaptivity is in the process management software included in the operating system of each node. In [Foster et al., 2000] a similar approach is used to provide network quality of service. The self-adaptivity is included in the network routers and it allows the system to efficiently use the network bandwidth. In both aforementioned examples the control system is composed of sensors, a set of decision procedures, and actuators. [Hafid and v. Bochmann, 1998] presents two protocols for QoS adaptation that allow to recover from QoS violations by

changing the distribution of QoS levels assigned to the network components in distributed multimedia applications.

Video coding and decoding applications have been the subject of a number of studies that investigate the control of quality and bitrate via application's parameters. Grant et al. [1997] investigate a fuzzy logic-based video rate controller aimed at regulating the data rate of compressed video at a constant transmission rate without objectionable quality degradation. In a similar effort, Rezaei et al. [2006] propose a fuzzy video rate controller designed for real-time variable bitrate applications with buffer constraints. Quality and bitrate are controlled by modifying the quantization scale. In [Cornbaz et al., 2005, 2007; Jaber et al., 2008] several techniques for fine-grained QoS control of multimedia applications are presented. Based on the estimates for the worst-case and average execution times for different levels of quality gathered by using timing analysis and profiling techniques, the proposed methods generate a controlled application that meets given QoS requirements from an input application software. The controller monitors the progress of the computation in a cycle and chooses the next action to run and its quality level, guided by safety and optimality constraints for the system.

Chapter 3

Reference Platform

In this chapter, we aim at providing the preliminaries of this thesis by describing the baseline platform on which we have implemented the fault tolerance and self-adaptivity techniques described in the following chapters. The baseline platform equipped with the architectural features described in section 3.1 is generated with the SHMPI builder tool [Meloni et al., 2010] by DIEE, Universitá degli Studi Cagliari. This choice derives from the MADNESS EU FP7 Project (No. 248424). The initial platform has been further developed by contributions from LIACS, University of Leiden and ALaRI, Faculty of Informatics, University of Lugano. The collaborative work leading to the creation of the reference platform, which includes the software support described in section 3.2, has been partially published in [Cannella et al., 2011; Derin, Diken and Fiorin, 2011; Derin and Diken, 2010]. Section 3.3 describes the overall fault tolerance approach adopted in the MADNESS project by detailing the fault model and the fault detection mechanism.

3.1 Architectural support

In this thesis, the system architecture is seen as a network of tiles, interconnected by means of an NoC communication infrastructure. In order to provide an actual experimental platform (rather than just a simulated one) we made use of a mesh-based 2×2 NoC multiprocessor architecture such as the one shown in figure 3.1 consisting of Microblaze processors implemented by means of a Xilinx Virtex6 FPGA; this choice does in no way detract from the generality of our solution. The methods and algorithms described in this work can be applied to NoCs with generic topologies. We consider an embedded system architecture composed of heterogeneous cores. Every tile contains a processing element and

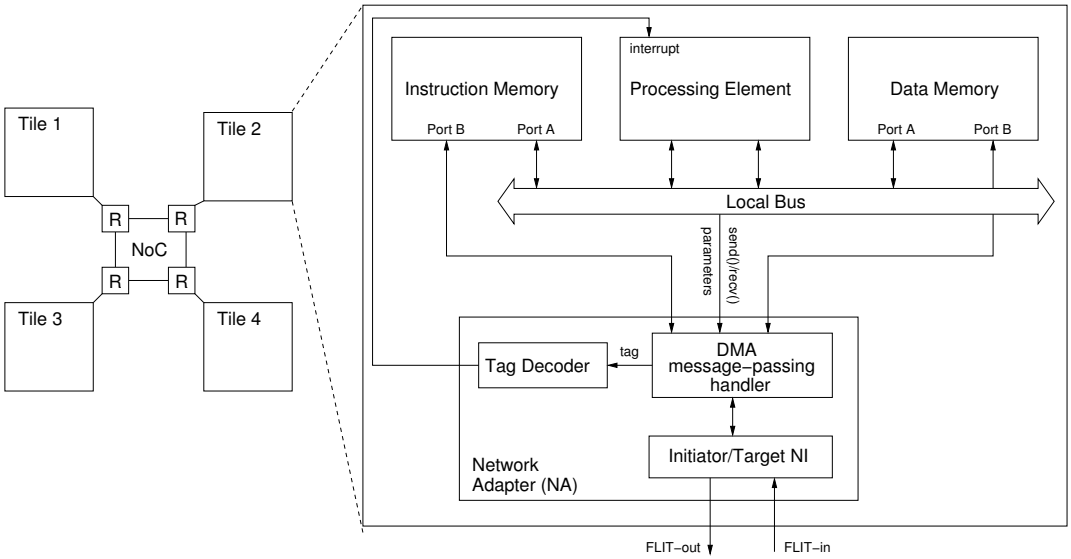


Figure 3.1. A general overview of the baseline tile architecture

its related local memory. Our platform implements pure message passing with the No Remote Memory Access (NORMA) model [Carara et al., 2007]. According to this model, tasks only have access to the local memory and there is no shared address space. Communication is performed through low level-message passing routines supported by the network interfaces of the NoC.

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 by accessing memory-mapped registers at the network interfaces. The template allows the connection of peripheral controllers that can be connected as network nodes and receive transactions initiated by processing elements. It is also possible to integrate non-programmable cores (i.e., custom processing elements) implementing specific dedicated algorithms, such as those used for audio or video encoding/decoding, or complex cryptographic functions.

The platform relies on a packet-switched source-based NoC, implementing a wormhole control flow. Without loss of generality, we consider in our experiments a XY routing algorithm. However, our formulation is valid for any deterministic routing scheme. Deterministic routing allows us to infer the communication binding from the task mapping (i.e., the task mapping is the only degree of freedom).

The communication network is built by using an extended version of the the ×pipes-lite library of synthesizable components [Dall'Osso et al., 2003]. 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 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 direct memory access (DMA) capabilities is integrated with the NI inside a module called *Network Adapter (NA)* described more in detail in section 3.1.1.

3.1.1 Message passing support

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. 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, respectively.

The NA architecture is depicted in figure 3.1.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)*, and a set of performance counters that keep 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 are 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 are 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.

3.1.2 Inter-processor interrupt generation support

A tag decoder has been instantiated inside the NA. It is in charge of detecting a set of predetermined 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. The number and the range of reserved tag configurations are

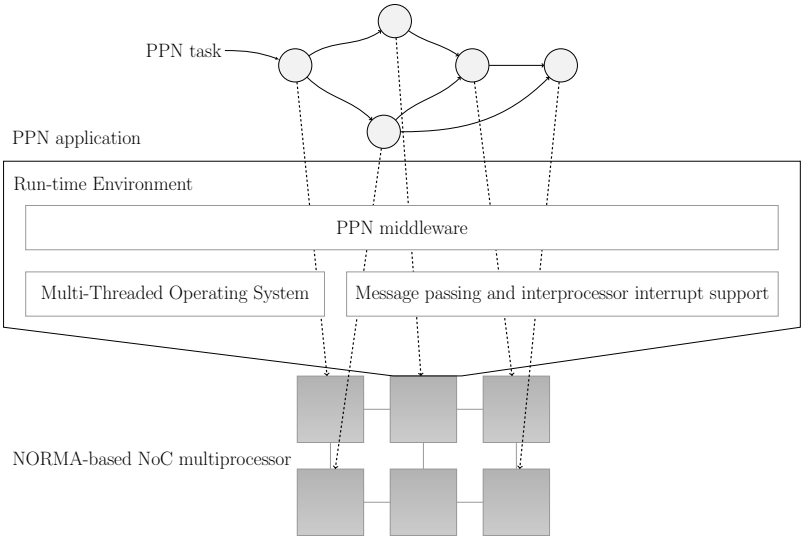


Figure 3.2. Software stack in the reference platform

configured at design time. By default, the tag is 16 bits wide, and 16 different configurations generate a different interrupt signal to the processor. This means that 16 comparators and 16 registers are instantiated inside the NA netlist after synthesis. If the interrupt signal is generated directly by the comparing logic, some spurious fluctuations can be generated at the receiving of the message, during the transient of the circuitry switching. Being the interrupt controller set to be sensitive on rising edge of the interrupt signal, in order to avoid such fluctuations to generate an interrupt in the processor, the interrupt signal is buffered in a register before being forwarded to the interrupt controller. This hardware overhead can be customized reducing the number of reserved tags according to the application features.

3.2 Software/Middleware infrastructure

Each tile of the system described in section 3.1 is endowed with the software/middleware stack depicted in figure 3.2. The *application level* resides at the top of the software stack. In the MADNESS project, applications are specified by using the PPN MoC. The application model is described below in section 3.2.1.

At the bottom of the software stack, the *multi-threaded operating system (MTOS)* provides basic functionalities such as process management (process creation/deletion, setting process priorities) and multitasking capabilities. We have used Xilkernel [Xilinx, 2010] and associated libraries as provided by Xilinx. The scheduler

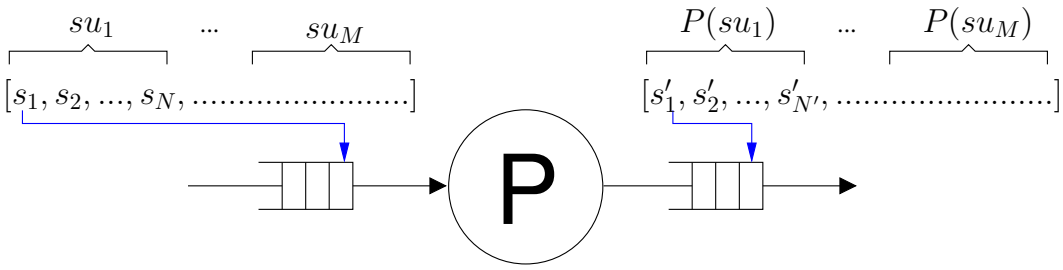


Figure 3.3. Example of a streaming application.

has been modified by University of Leiden to support a data-driven scheduling policy. When multiple processes are mapped on the same tile, the OS schedules the processes in a data-driven fashion. This means that a process runs until it blocks in reading/writing to/from a FIFO buffer. Then, it yields the processor control to the next process in the queue.

PPN middleware provides a communication API to the tasks by implementing the PPN semantics on top of the message passing support described previously in section 3.1.1. The details of PPN middleware are described in section 3.2.2.

3.2.1 Application model

A streaming application processes input data streams, possibly of infinite length, and produces output data streams. In this thesis (in particular in section 5.3 and chapter 6), we are interested in a specific type of streaming applications for which one can define *stream units* that can be processed independently by the application. Figure 3.3 exemplifies such an application where input stream units, su_i, are processed by the application, P, to produce the output stream units, $P(su_i)$. The application does not have a state resulting from previously processed stream units. Mathematically, for such applications, it holds that

$$P([su_1, ..., su_n, ...]) = [P(su_1), ..., P(su_n), ...]$$

Polyhedral Process Networks is a model of computation that is well suited to model such applications. In fact, such an application can be obtained as a composition of PPN processes such that each PPN process iterates over the stream units by means of their outermost loop. The beginning of the outermost loop (line 5 in figure 2.2(b)) corresponds to the execution point at which processing of a stream unit is finished, and the processing of the next stream unit has not been started. No state related to previous executions is stored in any process variable or any token in self-edges except the iterator value of the outermost loop. The

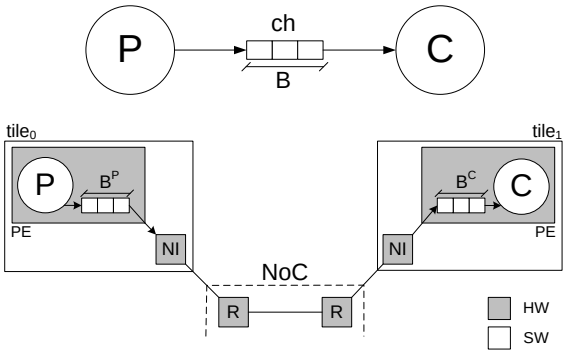


Figure 3.4. Producer-consumer pair with FIFO buffer split over two tiles.

fundamental advantage of using PPN is that this execution point is exposed for each process and exists for all applications conforming to our application model. This is the main enabler of the fault tolerance and self-adaptivity mechanisms described in the following chapters. It also facilitates separation of concerns by allowing the incorporation of programmatical changes to the PPN process structure in order to support fault tolerance and adaptivity.

3.2.2 PPN middleware

Based on the message passing support described in section 3.1.1, 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. Namely, the *blocking read* and *blocking write* primitives are implemented by the middleware.

The communication and synchronization problem when mapping PPNs on a NoC is depicted in figure 3.4. Consider a producer P and a consumer C connected through an asynchronous communication FIFO buffer B. If both the producer and the consumer can directly access the status register of this FIFO buffer, to check whether it is empty or full, implementing the PPN semantics is straightforward. However, in NoC implementations with no direct remote memory access, processes can exchange tokens only via the network. Thus, we have to split the buffer B in B^P and B^C, one on the producer tile and one on the consumer tile. We want to implement the PPN semantics without a dedicated support from the underlying architecture that allows checking for the status of the remote queues. If $size(B)$ is the minimum buffer size that guarantees deadlock-free execution of the original PPN graph, the size of B^P and B^C must be set such that

$size(B^P) + size(B^C) \geq size(B)$.

We do not require support for multiple hardware FIFOs on each NoC tile. The only hardware buffer of a tile resides in the NI. We just rely on the ability to transfer tokens, in both directions, from this buffer to the *software FIFOs* which implement the channels of our PPN.

Considering again figure 3.4, even if the consumer process C can only access the status of B^C, implementing the *blocking read* is trivial because every time process C wants to access B^C and this buffer is empty, the consumer just has to wait until tokens arrive from the producer tile. However, since the producer process B can only access the status of B^P, implementing the *blocking on write* behavior is more difficult. The producer must know that the remote buffer B^C is not full before sending tokens to C over the NoC. There are several ways to notify the producer about the status of the buffer on the consumer side.

Furthermore, we want the communication API to take care of the distribution of processes among the NoC tiles with no influence on the application designer. This means that we want to maintain the code structure of the PPN application processes, an example of which is shown in figure 2.2(b). In particular, we want the communication primitives (*read*, *write*) of PPN processes to remain generic, without the notion of process mapping or platform details. These generic primitives are then translated by the communication API implementation in mapping- and platform- dependent function calls.

System adaptivity in the form of online task remapping is taken into account by using dedicated middleware tables that list, among other information, the source and destination tile for each channel of the PPN graph. For instance, when a process is ready to send a packet to the consumer via a specific channel, the implementation of the *write* primitive will look up the current destination of that channel in the middleware table. Then, it will place the packet in the NI output buffer, with the appropriate destination field of the header. As described in chapter 5 , these middleware tables are updated at run-time to allow online remapping of application processes over the tiles as a part of the fault recovery mechanisms.

We describe several methods to implement the PPN communication over NoC-based MPSoCs in [Cannella et al., 2011], namely *Virtual Connector, Virtual Connector with Variable Rate, and Request-driven*. The *Request-driven* communication approach is adopted as it leads to an easier implementation of the fault recovery mechanism, thanks to the reduced number of synchronization points between the processes in this approach.

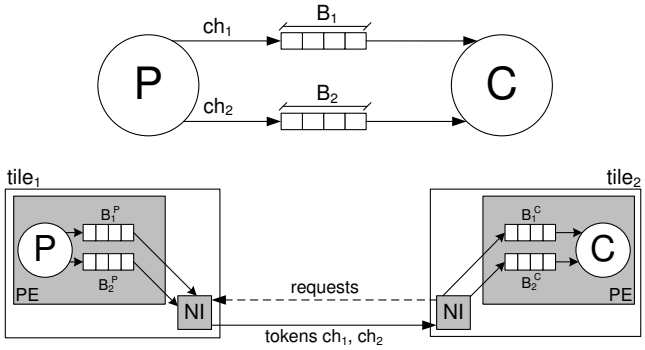


Figure 3.5. Request-driven inter-tile communication implementation

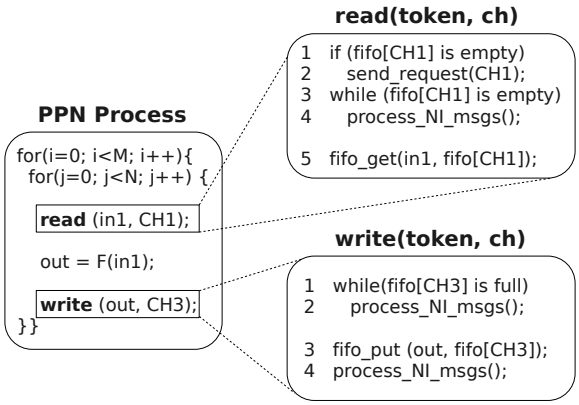


Figure 3.6. Pseudocode of the R approach.

Request-driven approach (R)

This method is similar to the approach used in Nadezhkin et al. [2009] for realizing the FIFO communication on the Cell BE platform. In this approach, 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.

Since also in this case we need to store tokens both on the producer side and on the consumer side, we need software FIFO structures on both sides. The size of these buffers is set, for each channel i, to match the size of the queue in the original PPN graph (B_i), such that for all channels B_i in the original graph, $B_i^P = B_i^C = B_i$. This condition guarantees deadlock-free execution on the NoC. The structure of a producer-consumer pair using the R approach is shown in

figure 3.5. Since the consumer buffer of a channel is empty when a request is made, and given that the FIFO buffers for that channel have the same size on both sides, there is always enough space to store tokens sent by the producer as a consequence of the request.

Figure 3.6 shows the pseudocode of this communication approach. Both the *read* and *write* primitives use an auxiliary function, *process_NI_msgs()*, that is used when blocking on read or on write. This function checks the status of the NI buffer for incoming packets. If the buffer is not empty, this function processes one packet at a time, until all the incoming packets are consumed, in the following way. If the packet is an incoming token for channel i, it stores the token in the software FIFO which implements channel i. If it is a request message for channel j, it sends immediately all the tokens contained in the software FIFO that implements channel j.

The *blocking on read* behavior is implemented in lines 1-4 of the read primitive in figure 3.6. When the software FIFO of the calling channel is empty, a request is sent to the producer tile of that channel, and the processor keeps executing *process_NI_msgs()* until a packet of tokens for the calling channel arrives. The *blocking on write* is implemented in lines 1-2 of the write primitive in figure 3.6. When the FIFO of the calling channel (in the example, *CH3*) is full, the processor keeps executing *process_NI_msgs()* until a request for that channel arrives.

3.3 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*. 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. In fact, the work done on fault-tolerant NoCs within the MADNESS project [Fiorin and Sami, 2013] or elsewhere [DeOrio et al., 2012] achieve designs that can survive a high number of faults in the interconnect with considerably small area overhead. Therefore, in this work, the NoC is considered to have a much longer lifetime than processing elements and still to be able to guarantee reliable communication services even in the case in which one or more processing

elements stop working correctly. Memories are assumed to be protected against faults by employing standard fault tolerant techniques based on the use of Error Correcting and Detecting codes [Koren and Krishna, 2007].

Fault tolerance consists of fault detection and recovery phases. In this thesis, we describe the work done to enable continuity of service by recovering from permanent faults in processing elements when running PPN applications on NoC multiprocessors. The fault model is detailed in section 3.3.1. We assume the presence of a fault detection mechanism that makes use of online software-based self-testing [Gizopoulos, April-June 2009] and the self-testing module as described in section 3.3.2.

3.3.1 Fault model

With regard to the fault model adopted, this work considers errors in a processing element derived from permanent faults (either logic or delay faults occurring in the combinational logic or registers of processing elements). Adopting as level of abstraction the processor-memory-interconnect level, we adopt the single fault assumption, that is, one processing element at a time can fail and that recovery is completed before possibly a new fault appears in another processing element. If a processing element is found by the fault detection mechanism to produce incorrect results, it is excluded from the normal system activity and the recovery mechanism is invoked.

Traditionally fault tolerance techniques for processing elements have been employed in mission critical systems such as space missions or highly available applications [Gaisler and Catovic, 2006; Reick et al., 2008]. These are mostly hardware techniques at micro-architectural or gate level. The handling of the fault is not escalated to the software level. In the context of non-mission critical applications that make use of standard processing elements, the main philosophy of our approach is exploiting the redundant processing power available in the multi-core platform in order to compensate for the absence of faulty processors by leveraging run-time techniques in software.

Rather than relying on fault-tolerant hardware, cross-layer approaches embrace faulty hardware and allow the elevation of errors to the software level. There has been increasing attention to understand the interplay between hardware faults and application-level errors. Such application-specific hardening techniques, which have focused mostly on transient faults, adopt a top-down [Schmoll et al., 2013] or a bottom-up [Yetim et al., 2013] approach. The former investigates the susceptibility of the application to errors and aims at identifying critical parts of the application and hardening them, while the latter evaluates

the impact of hardware faults on the application by means of a fault injection campaign and aims at identifying critical parts of the hardware to be hardened to minimize application-level errors.

Fault injection is an important design activity that helps finding out the classes of error that can be observed and gives statistically a rate of their occurrence. Such an analysis needs to be done for every processor type in the platform and can be a tedious and complicated task even for a simple processor. Moreover, it requires one to have at hand the design of the processor as a netlist, which is not the case, in particular, for the proprietary Microblaze processor that have been used in our reference platform and more in general for many units proposed as IPs to be inserted in complex Systems-on-Chip (SoCs). Therefore we refrained from doing the analysis of the fault and error models at the gate level. Instead we refer to the insights obtained by previous work [Li et al., 2008; Pellegrini et al., 2012]. Pellegrini et al. [2012] investigated the impact of permanent stuck-at and path-delay faults on five SPECInt 2000 benchmark applications via 50,000 fault injection experiments carried out on the OpenSPARC processor. Possible outcomes of system's behavior have been categorized and measured as follows: masked (60.7% for stuck-at, 65.3% for path-delay), detected via software anomaly detectors (30.4% for stuck-at, 29.4% for path-delay), time-out mostly due to hanging (7.8% for stuck-at, 4.2% for path-delay), silent data corruptions (0.82% for stuck-at, 0.75% for path-delay) and other anomalous outcomes (0.2% for stuck-at and 0.4% for path-delay). In the case of silent data corruptions (SDCs), the application finishes but its output differs from the expected one. Authors note that a vast majority of SDCs are concentrated in units that compute data values for the program such as floating point unit, multiplier, divider (in particular, floating point unit had the highest SDC rate at 7.55% for stuck-at faults and 10.47% for path-delay faults).

As the above results reveal, 80.2% of the unmasked faults can be detected by software anomaly detectors. Some of the anomaly types are hardware fatal traps, kernel panics and application aborts. Since these are detected at software level, a recovery process can be initiated by signaling the fault to the fault handler. If one has to handle remaining faults as well, SDCs ($\sim$2% of the unmasked faults) and time-outs ($\sim$16% of the unmasked faults) which are representative of processor hangs (e.g., due to infinite loops or stuck program counter) should also be considered.

In our setting where the processor is not alone but exchanges messages with others, hangs and SDCs can be elaborated further. First, we take a top-down approach and analyze errors from the PPN application's perspective. We define two main error classes as the following:

- E1: application is blocked
  1. process hangs (e.g., due to an infinite loop caused by wrong evaluation of a conditional branch)
  2. there is an incorrect number of tokens in the FIFOs due to abnormal read / write rates in the channels (e.g., a process blocks forever on reading an empty channel or writing to a full channel)
  3. a process is blocked reading from a non-existing channel
  4. a process is blocked writing to a non-existing channel
- E2: application continues with unlimited error propagation
  1. the number of tokens in FIFOs is correct but the associated values are wrong

In case E2.1, thanks to the PPN model, the application will not be blocked due to wrong token values because they do not determine the control flow. If that was the case, for example as in dynamic dataflow models such as KPN, it might have been possible for the corrupt data to be used by the successor task to define its iteration boundaries. Then, the successor task might have ended up executing a greater or lower number of iterations and producing incorrect computation results. In the former case, the successor task would have eventually blocked waiting for more data while, in the latter case, it would have finished early with unconsumed data in the input queues.

We take then a bottom-up approach and analyze the corruptions that may lead to the above defined error classes. Note that these corruptions occur with small probability as most faults exhibit more relevant symptoms detectable by aforementioned software anomaly detectors [Pellegrini et al., 2012].

- C1: process-level corruptions
  1. *corruptions in process function*: When running the process function, a permanent fault may lead to an SDC resulting in the wrong calculation of a token value (leading to E2.1). The application output will be corrupt (e.g., incorrect pixel values will appear in a video application) as long as the fault is not handled.
  2. *corruptions in read or write*: read and write operations are affected by corruptions in the middleware level and FIFO operations (see below). In addition to them, a fault may cause reading from a wrong channel which will lead to E1.3 or E1.2 (as discussed in C2.1). Similarly, it

may cause writing to a wrong channel which will lead to E1.4 (as discussed in C2.5).

3. *corruptions in FIFO operations*: get and put operations on the FIFO data structure may be corrupt in several ways. On the one hand, similar to C1.1, a wrong token value may be written in the FIFO buffer due to a corrupt argument (leading to E2.1). On the other hand, other variables of a FIFO such as read pointer, write pointer and size may be corrupt effectively resulting in a FIFO with smaller capacity or lost tokens. This will block execution (leading to E1.2).

4. *corruptions in the iteration construct*: The process body may execute a wrong number of iterations due to corrupt iterator values or bounds, or due to wrong evaluation of a conditional branch. For instance, the process may break its main loop early leading to a similar situation as blocking of the process. Since iterators and iterator bounds capture the control flow of the application, their corruption will most likely block the application either causing an infinite loop (E1.1) or changing the read/write patterns of the process (E1.2) by corrupting the target/source domains of the input/output channels.

- C2: middleware-level corruptions

1. *wrong value of the receive tag*: If the wrong value of the tag does not correspond to an existing channel, the PE will be blocked forever leading to E1.3 (since FIFO channels are defined by tags). If the wrong receive tag happens to be a value of an existing channel, then the tokens destined for a different channel and currently residing in the NI buffer will be transfered to a wrong FIFO. The task will be consuming data that is not intended for it. Producer of the actual channel that should have been read will be blocked on write and the actual consumer of the wrong channel will be blocked on read forever (leading to E1.2).

2. *wrong value of the receive processor ID*: The faulty PE will be blocked forever waiting for an input from a wrong processor (leading to E1.3).

3. *wrong value of the memory location where the received message is stored*: The received message may be copied to a wrong memory location corrupting the data in that location. The process will continue as if new tokens had been copied in the FIFO buffer. Since that is not the case, it will continue with wrong token values that remained in

memory from earlier tokens (leading to E2.1). Alternatively, trying to access a wrong memory location may raise a segmentation fault causing the process to abort (leading effectively to E1.1).

4. *wrong value of the size of the message to be received*: If the value of size is smaller than it should be, then a shorter message will be received which leads to writing an incomplete token value in the FIFO buffer. This translates as a corrupt token similar to case C1.1, thus leading to E2.1. If the value of size is bigger, then it may practically mean that the receiving task is blocked until that many bytes are received on that specific channel. However this may create a deadlock (leading to E1.2).

5. *wrong value of the send tag*: The faulty PE will send the message to a wrong and possibly non-existent channel. The successor task will be blocked because it will never receive a message on the expected channel. The producer task on the faulty PE will also block due to a full channel that is never read by a consumer task (leading to E1.4).

6. *wrong value of send processor ID*: The faulty PE will send the mesage to a wrong processor. The successor task will be blocked because it will never receive a message. The producer task on the faulty PE will also block due to a full channel that is never read by a consumer task (leading to E1.4).

7. *wrong value of the memory location of the message to be sent*: The faulty PE will send a wrong message, that is, corrupt tokens. It will lead to E2.1 similar to case C1.1.

8. *wrong value of the size of the message to be sent*: A shorter or longer message will be sent over the network. In the case that the sent message is shorter, then the successor task will block because it will not have received a properly sized token (leading effectively to E1.2). In case the sent message is longer, then there may not be enough space in the destination tile's NI buffer to store the message, which will create an overflow error in destination tile's NI. Even if the extra long message is stored successfully in the destination, the extra portion of the message will be read as wrong tokens in future iterations of the successor task, thus it will lead to E2.1.

9. *corruption in requests*: Corruption in a request may cause either an appearance or a disappearance of a request on a channel. The former case is harmless, the request will be served by sending the tokens

of the channel if there are any. However, the latter will cause the successor task to block forever on read as it will never receive any tokens due to the request never being served (leading to E1.2).

- C3: OS-level corruptions

  The scheduler may fail to schedule a task for execution even though it should not be blocked on read or write, possibly due to a corruption in yielding or in the priority levels of the processes (leading to E1.1).

An application interacts with its environment (outside world process) by reading input data and producing output data. Ideally a fault tolerant system should not deliver a wrong output even in the presence of faults. The problem of making sure that the data do not contain any error is called the output commit problem [Sorin, 2009]. Although the BER techniques described in section 2.8.2 allow to remove internal errors, the error that has propagated to the outside world cannot be undone (e.g., errors on the paper printed by a printer, errors in images displayed by a monitor). The common solution to this problem is to buffer a sufficient amount of the input and output data at the expense of additional memory so that they can be rolled back during recovery [Nakano et al., 2006; Sastry Hari et al., 2009]. A very important constraint in our platform is the memory size. The free space in the memory of each tile is not large enough to store a whole stream unit. This limitation is fundamental for the fault recovery mechanisms proposed in this thesis. Given the results reported by the checkpointing and rollback techniques in [Constantinides et al., 2007; Sastry Hari et al., 2009], hundreds of kilobytes of memory requirement per core for logging purposes is not feasible in our platform which provides only 64 KB of data cache per core. Furthermore we relax the output commit problem and allow limited error propagation due to the nature of the multimedia applications that are adopted as use case applications.

3.3.2 Online self-testing support

Although this thesis focuses on the recovery problem rather than detection, we provide some details on the fault detection support. If the application is not critical and a limited amount of error propagation is acceptable, a self-testing routine can be executed by the processing element to detect its permanent faults [Gizopoulos et al., 2008; Foutris et al., 2010; Scholzel et al., 2012]. Due to the costs associated with concurrent checking techniques, we adopt online software-based self-testing. However the recovery techniques proposed in this thesis can be used in combination with other fault detection techniques. More specifically,

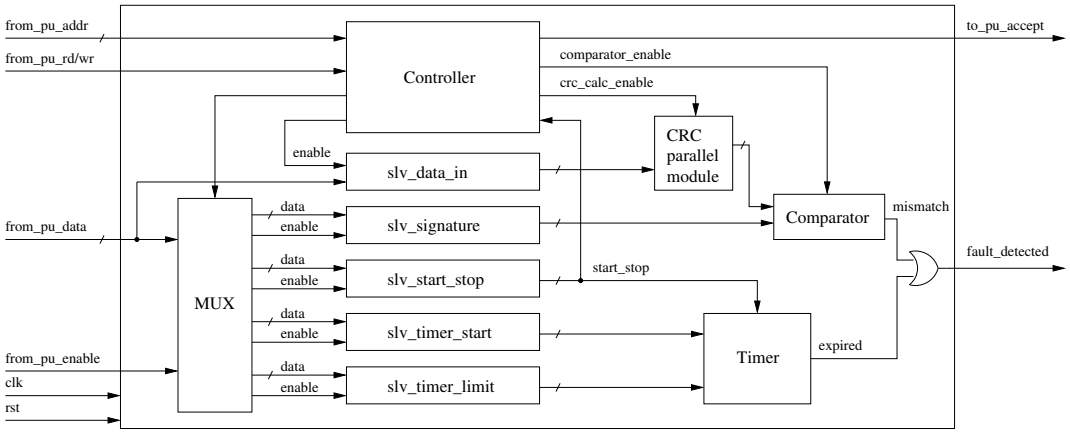


Figure 3.7. Overview of the STM architecture

the rollback based recovery technique proposed in section 5.2 can be used in combination with other detection mechanisms (e.g., [Ananthanarayan et al., 2013]) that have short detection latencies (in comparison to the duration of one iteration of a task) so that errors do not escape the faulty core. On the contrary, the roll-forward based recovery technique proposed in section 5.3 does not have such a constraint.

A hardware unit is needed in each tile to help with self-testing. The *Self-testing Module (STM)* checks the results of the software testing routines and signals the detection of a fault. The STM is in charge of collecting the outputs of the processor when executing the software routine, calculating the signature of the outputs of the processor, and checking it against the expected signature, in order to verify the correctness of the routine execution. The signature is calculated by applying a cyclic redundancy check (CRC) algorithm to the obtained outputs of the processor [Koren and Krishna, 2007]. The testing routine, as well as the signature of the expected results, are stored directly in the processor local memory. Results of the execution of the software routine are written directly into the STM.

Figure 3.7 shows the architecture designed within the MADNESS Project 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 (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 do not match, the *fault_detected* signal is set to '1' for a clock cycle. The STM also supports detecting hang errors (e.g., halting of the program counter or blocking forever at an execution point). Such errors would result in the processor not executing the self-testing routine, which is executed frequently under normal conditions. A watchdog timer is introduced inside the STM to detect such errors. If the self-testing routine completes (inferred by a write into the *slv_start_stop* register) within a time limit defined by *slv_timer_limit*, the timer is reset. Otherwise, a hang error is assumed and the *fault_detected* signal is raised. The time limit should be chosen according to the workload of the processing node.

Although the self-testing module has been designed within the MADNESS Project by other colleagues, it was not integrated into the platform at the time the fault recovery experiments in chapter 5 had been performed. Therefore, when carrying out the experiments, the *fault_detected* signal is raised directly by setting a memory-mapped register in the process body.

Chapter 4

Fault-aware Online Task Remapping

Fault-aware online task remapping deals with finding the new processing nodes where the KPN tasks shall continue their execution upon detection of a permanent fault in a processor. The main problems tackled in this chapter are the remapping strategy that finds the new nodes to be allocated for the tasks running on the faulty node and the impact of the fault-aware online task remapping approach on the lifetime reliability of the system. The results presented in this chapter have been published partially in [Derin, Kabakci and Fiorin, 2011; Derin, Cannella, Tuveri, Meloni, Stefanov, Fiorin, Raffo and Sami, 2013; Derin and Fiorin, 2014].

The remainder of this chapter is organized as follows. Section 4.1 discusses the contributions of this dissertation with respect to the state-of-the-art. Section 4.2 presents the ILP formulation for the mapping problem based on an analytical model for throughput and communication cost. Section 4.3 extends the ILP formulation for the remapping problem and also proposes a set of heuristics for online task remapping. Section 4.4 proposes an analytical model for estimating the lifetime reliability of a system that adopts the fault-aware online task remapping technique. N-modular redundancy technique is also considered for comparison purposes. Section 4.5 presents several case studies and results obtained both analytically and experimentally on the actual platform.

4.1 Contributions with respect to the state of the art

The fault-aware online task remapping problem is closely related to the task mapping problem. The task mapping solutions proposed by the related work overviewed in sections 2.6 and 2.8.3 have some shortcomings.

Most approaches deal with application-specific synthesis of a NoC-based system. The final system is designed and optimized to execute a specific application. Since this problem involves designing also the hardware platform, the problem is divided into several steps such as partitioning, scheduling, mapping, routing, topology generation etc. Even if each of the individual sub-problems can be solved optimally, a globally optimal solution cannot be guaranteed with such an approach.

The problem dealt with in this work adopts the platform-based design approach. Rather than synthesizing a new hardware system for the application, an application is mapped onto an existing platform possibly by some customizations and tailoring. Choosing among the platform-based design and platform synthesis approaches is not just a matter of preference. The decision is influenced by the availability of a platform for the application at hand and, to a larger extent, by the expected volume of sales due to the current costs of manufacturing semiconductor products. Although the adopted platform may not be the ideal platform for the application, platform-based design enables economically viable solutions. TI OMAP [Cumming, 2003], NXP Nexperia [Oliveira and van Antwerpen, 2003] and ST Platform 2012 [Melpignano et al., 2012] are some platform examples designed for a wide range of embedded multimedia applications.

When adopting the platform-based design approach, the design problem is reduced to mapping and scheduling the application tasks on a NoC-based platform that has a number of IP cores mapped already on the nodes of the NoC. The optimization of computation and communication, which is done in two steps consisting of partitioning and core mapping, is merged into a single task mapping step, thus allowing the simultaneous optimization of both metrics.

In our endeavour for a fault-aware online task remapping solution, firstly, we present a method for finding an optimal solution to the task mapping problem aiming at maximizing application throughput (by minimizing the total execution time) and minimizing the communication volume on the network, given NoC-based platforms with generic topologies and deterministic routing algorithms. Then, the optimal mapping method is extended for the task remapping problem. Such an optimal solution constitutes the basis for assessing the quality of the online remapping methods that are proposed.

Within the context of KPN applications, a simple but accurate analytical model is adopted which can estimate the metrics of interest given the mapping on a NoC-based platform. The optimization is based on the analytical model rather than measurements via simulations or directly by emulations of the platform due to the fundamental reason that the online remapping decisions are taken by heuristic methods. Simplicity of the analytical model also leads to lightweight

heuristics. The alternative approach (e.g. [Lee et al., 2010]), which would make optimal remapping decisions based on an offline (design-time) analysis may provide more predictable and faster recovery times, but it has some drawbacks. First, it would require a considerable amount of effort to evaluate all fault scenarios. Second, it would require a large memory overhead for storing the remapping decisions to be used at run-time. Finally, the actual working conditions of the system may be different than those used during the offline analysis (e.g. the processor performance may change due to frequency scaling or the running application may be different) which would lead to non-optimal decisions.

In the case of synthesis-based flows, which employ fully static scheduling, a possible show-stopper for achieving fault tolerance by remapping tasks is the re-scheduling problem. The static schedule represents the processor assignments, iteration period and firing times of each task. When a remapping is needed due to a fault occurrence, determing the new processor assignment as well as computing the new schedule, which are heavy computations even at design time, would cause a large run-time overhead and result in a schedule far from optimal. Adopting a dynamic scheduling policy such as the data-driven scheduling as in our case, the re-scheduling problem reduces to a remapping problem. According to data-driven scheduling, a KPN task is executed until it is blocked on a read or a write. Another task which has input data available to process is scheduled next. The analytical model used in our solution is valid for such a scheduling policy.

As mentioned in section 2.8.3, one of the approaches for fault-aware mapping is using spare cores. In this approach, application components running on a faulty core are migrated as a whole to one of the available unemployed spare cores. This approach is suitable for the application-specific synthesis flow described previously. It is an extension to the core mapping problem where a core, which refers to scheduled tasks bound to a core type, is mapped to a node of the NoC. In this approach, same tasks are simply run with the same schedule on a spare core after the remapping. The solutions that adopt such an approach mainly deal with optimizing communication by deciding on the placement of spare cores and the selection of the spare core for the remapping [Ababei and Katti, 2009; Chou and Marculescu, 2011]. Our work is fundamentally different in that the remapping is done on a task basis, i.e., each task can be remapped on a different processing node. Although the computational overhead becomes not equally predictable, the cost of adding unemployed spare cores is avoided; this is a basic aspect in our case, since cost is in general a strict constraint for embedded systems. Remapping at the granularity of tasks allows better utilization of the platform resources than core-level granularity.

We also investigate the reliability aspect of the fault-aware online task remapping technique by looking at its impact on the lifetime of the system. Previous work [Huang et al., 2009; Huang and Xu, 2010; Huang et al., 2011] sought to achieve a similar goal by estimating the lifetime of real-time systems. In such systems, the scheduling of tasks running on a processor is fixed. Therefore the thermal profile of the processor, which represents the change of temperature in time, can be obtained which would be then fed into a wear-out model to obtain an MTTF value for the processor. This is unfortunately not applicable when dealing with throughput-oriented systems using the data-driven scheduling policy as in our case. Morever, in the fault-aware remapping scenarios, the tasks running on a core are dynamic unlike the static mapping assumed by the mentioned previous work.

Since our focus is on the use of software based solutions for fault tolerance, we adapt the *N-modular redundancy* (NMR) [Koren and Krishna, 2007] technique to be used at the application level. Similarly, we propose a method to estimate the MTTF of NMR-ed applications and compare it against the online remapping technique.

Design space exploration relies fundamentally on an exploration engine and on estimators that are able to evaluate a design point with regard to the metrics of interest. Given an application and its optimization goals, the exploration engine identifies the designs that better satisfy the goals. A reliability-aware DSE is able to employ a reliability metric as a goal in the design. Fault tolerance techniques such as OTR and NMR expose to DSE tools a new dimension to be explored. The reliability estimation methods proposed for OTR and NMR enable such a DSE.

Our contributions in this field are

- defining an analytical model for estimating performance metrics (i.e., execution time and communication cost) of KPN applications running on NoC-based platforms,

- proposing a method for optimal mapping and remapping of KPN applications on NoCs,

- proposing online remapping heuristics reacting to run-time faults that minimize degradation,

- defining an analytical model for estimating the lifetime reliability achieved by means of the fault-aware online task remapping and N-modular redundancy techniques,

- evaluating the quality of the heuristics by comparing the results against the optimal ones,
- validating the quality of the heuristics by measurements on an actual NoC-based platform,
- evaluating the calculation time of the heuristics on an actual NoC-based platform,
- evaluating the impact of the remapping technique on the lifetime reliability of the system in comparison to NMR.

4.2 ILP formulation of the mapping problem

For convenience, table 4.1 summarizes the basic notation that will be used hereafter. With these notations, the problem presented in this paper can be described by the following elements:

- a *task graph* $g_t = (V_t, E_t)$ is composed of tasks $t \in V_t$ and data dependencies $e \in E_t \subseteq V_t \times V_t$;
- an *architecture graph* $g_a = (V_a, E_a)$ is composed of processing nodes $n \in V_a$ and bidirectional communication links $l \in E_a \subseteq V_a \times V_a$;
- a *task mapping* function $\beta(t) : V_t \rightarrow V_a$ is an assignment of tasks $t \in V_t$ to nodes $n \in V_a$;
- a *data dependency mapping* function $\beta(e) : E_t \rightarrow E_a^i$ is an assignment of data dependencies $e \in E_t$ to paths of length i in the architecture graph g_a. A *path* p of length i is given by i-tuple $p = (l_1, l_2, ..., l_i)$;
- $path : (E_a, E_a) \rightarrow E_a^i$ is a function that implements a deterministic routing algorithm and returns a path between two given nodes. *Path set* P is the set of paths between all node pairs:

$$P = \{p_k : p_k = path(n_i, n_j), \forall n_i, n_j \in V_a \wedge n_i \neq n_j\} \tag{4.1}$$

  Initial and final nodes of a path can be obtained by *source* and *sink* functions.

$$p_k = path(n_i, n_j) \Rightarrow source(p_k) = n_i \wedge sink(p_k) = n_j \tag{4.2}$$

Table 4.1. Table of notations

Symbol	Meaning
g_t	task graph
V_t	task set
E_t	data dependency set
g_a	architecture graph
V_a	processing node set
E_a	communication link set
$\beta(t)$	task mapping
$\beta(c)$	communication binding
P	path set
d_i	required bandwidth between two tasks
l_i	bandwidth between two nodes
C	types of core available in the platform
X^{NT}	incidence matrix denoting the mapping of tasks onto the nodes
Y^{PE}	incidence matrix denoting the mapping of data dependencies onto the paths
M^{TE}	oriented incidence matrix relating tasks to data dependencies
M^{NP}	oriented incidence matrix denoting the relation between paths and nodes
M^{PL}	incidence matrix denoting relation between all paths and the composing links
M^{TC}_{cap}	incidence matrix denoting which core can execute a specific task
T^{TC}_{cap}	matrix denoting the completion time of all tasks on all core types
M^{NC}	incidence matrix denoting the core type of a node
T^N	vector denoting the sum of execution times of the tasks on a node

- the task graph can be annotated with demand values where *demand* d_i on a data dependency $e_i \in E_t$, denotes the required bandwidth between the two tasks. Demand values are application specific and can be calculated by profiling the application with a test input;

- the architecture graph can be annotated with capacity values where *capac-*

ity on an architectural link $l_i \in E_a$, c_i, denotes the maximum bandwidth of the communication link between two architectural nodes;

- core type set C consists of core types C_i and lists the types of cores available in a given NoC platform.

We aim at minimizing the total network traffic and the computation time.

4.2.1 Minimization of the communication cost

In order to formulate the problem, we define several incidence matrices, namely the ones related to decision variables X^{NT}, Y^{PE}; and parameters M^{TE}, M^{NP} and M^{PL}.

X^{NT} is an incidence matrix of size $|V_a| \times |V_t|$ that denotes the mapping of tasks onto the nodes and it consists of the main decision variables of the problem.

$$X_{ij}^{NT} = \begin{cases} 1, & \text{if } t_j \in V_t \text{ is bound onto node } n_i \in V_a \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Y^{PE} is an incidence matrix of size $|P| \times |E_t|$ that denotes which path realizes which data dependency. Y^{PE} depends on the task mapping, hence it constitutes the second set of our decision variables.

$$Y_{ij}^{PE} = \begin{cases} 1, & \text{if } e_j \in E_t \text{ is mapped to } p_i \in P \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

M^{TE} is an oriented incidence matrix of size $|V_t| \times |E_t|$ that relates the tasks to the data dependencies. For a given task graph, M^{TE} is known.

$$M_{ij}^{TE} = \begin{cases} 1, & \text{if } \exists t_k, e_j = (t_i, t_k) \in E_t \wedge i \neq k \\ -1, & \text{if } \exists t_k, e_j = (t_k, t_i) \in E_t \wedge i \neq k \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

M^{NP} is an oriented incidence matrix of size $|V_a| \times |P|$ that denotes the relation between the paths and the nodes that the path connects. For a given routing algorithm and architecture graph, M^{NP} is known.

$$M_{ij}^{NP} = \begin{cases} 1, & \text{if } source(p_j) = n_i \\ -1, & \text{if } sink(p_j) = n_i \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

M^{PL} is an incidence matrix of size $|P| \times |E_a|$ that denotes the relation between all paths resulting from a given deterministic routing algorithm and the links that

make up the path. For a given routing algorithm and architecture graph, M^{PL} is known.

$$M_{ij}^{PL} = \begin{cases} 1, & \text{if } l_j \in p_i \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

It is to be noted that the transpose of the incidence matrices are expressed by swapping the letters that appear in their superscripts. For example, $X^{TN} = (X^{NT})^T$, $Y^{EP} = (Y^{PE})^T$, $M^{LP} = (M^{PL})^T$.

Constraint 1 *(routing)*: we have derived the following linear equation that constrains the task mapping and the communication binding with each other. Such a constraint arises from the deterministic routing algorithm implemented in the NoC.

$$X^{NT} M^{TE} = M^{NP} Y^{PE} \tag{4.8}$$

Constraint 2 *(task mapping)*: a task can be mapped exactly on one node.

$$X^{TN} \mathbf{1}_{|V_a|} = \mathbf{1}_{|V_t|} \tag{4.9}$$

where $\mathbf{1}_m$ is a matrix of size $m \times 1$ with all elements equal to 1.

Constraint 3 *(data dependency mapping)*: a data dependency can be mapped at most on one path.

$$Y^{EP} \mathbf{1}_{|P|} \leq \mathbf{1}_{|E_t|} \tag{4.10}$$

Constraint 4 *(capacity)*: total bandwidth demand on a link l_j should not exceed the capacity of the link c_j.

$$M^{LP} Y^{PE} d \leq c \tag{4.11}$$

Objective 1 *(minimization of the communication cost)*: the total traffic on the links can be calculated as the sum of all demands d_i on the links of the paths that arise according to a given mapping with the following equation.

$$\text{Minimize } d^T \, Y^{EP} M^{PL} \, \mathbf{1}_{|E_a|} \tag{4.12}$$

This is a static model that has also been used in [Murali and De Micheli, 2004] and disregards the congestion on the links. However, at low load conditions, it is argued that it is a good approximation.

Note that the communication cost takes into account the inter-tile communication done over the NoC between tasks and not the intra-tile communication when communicating tasks are mapped onto the same node. The latter is usually much faster compared to the former.

Therefore, the objective for communication is the minimization of the total traffic (equation 4.12) subject to routing algorithm constraints (equation 4.8),

mapping constraints (equation 4.9, 4.10) and capacity constraints (equation 4.11). Since the equations are linear, this problem can be solved with an integer linear programming (ILP) solver.

Given our analytical cost model, it is obvious that when communication cost is taken as the only objective, the resulting mapping will always be that all tasks are mapped on a single node. However, the low parallelism produced by this solution will reflect badly on the computation time. Therefore, we introduce a conflicting second objective that favors tasks to be placed on separate nodes.

4.2.2 Minimization of the total execution time

Application throughput, as a measure of performance, is the amount of data processed over a period of time. Minimization of the total execution time maximizes the application throughput. In order to formulate this objective we define additional parameters in matrix form, namely M^{TC}_{cap}, T^{TC}_{cap} and M^{NC}.

M^{TC}_{cap} is an incidence matrix of size $|V_t| \times |C|$ that denotes which core types are capable of realizing which tasks. Programmable cores would be capable of realizing different kinds of task functionalities, whereas non-programmable cores would have dedicated functions.

$$M^{TC}_{cap_{ij}} = \begin{cases} 1, & \text{if } t_i \in V_t \text{ can be realized by } C_j \in C \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

T^{TC}_{cap} is a matrix of size $|V_t| \times |C|$ that denotes the computation time of all tasks on all core types for a test input. This value is obtained by multiplying the number of times the task body is executed by the time it takes to process at each firing. Given an application and architecture, this matrix can be obtained by offline profiling.

$$T^{TC}_{cap_{ij}} = \begin{cases} \text{computation time of } t_i \text{ on } C_j, & \text{if } M^{TC}_{cap_{ij}} = 1 \\ 0, & \text{if } M^{TC}_{cap_{ij}} = 0 \end{cases} \tag{4.14}$$

M^{NC} is an incidence matrix of size $|V_a| \times |C|$ that denotes the core type of the architectural nodes. Given an architecture, M^{NC} is known.

$$M^{NC}_{ij} = \begin{cases} 1, & \text{if } n_i \in V_a \text{ is of core type } C_j \in C \\ 0, & \text{otherwise} \end{cases} \tag{4.15}$$

The execution time $T^{NT}_{cap\,ij}$ of task t_j if assigned to node n_i can be calculated in matrix form as

$$T^{NT}_{cap} = M^{NC} T^{CT}_{cap} \tag{4.16}$$

T^N is a vector of size $|V_a| \times 1$. T_i^N denotes the sum of execution times of tasks that are mapped on the same node, n_i. It can be calculated as

$$T^N = (T_{cap}^{NT} \cdot X^{NT})\ \mathbf{1}_{|V_t|} \tag{4.17}$$

where the dot(·) operator represents element-wise matrix multiplication.

Constraint 5 *(capability)*: all tasks should be mapped on cores that are capable of implementing those tasks.

$$M^{TC} = X^{TN} M^{NC} \leq M_{cap}^{TC} \tag{4.18}$$

Objective 2 *(minimization of the total execution time)*: we calculate the total execution time of the application by finding the maximum of the sum of the execution times of tasks mapped on the same core.

$$\text{Minimize } max(T^N) = max((T_{cap}^{NT} \cdot X^{NT})\ \mathbf{1}_{|V_t|}) \tag{4.19}$$

where max is a function that returns the maximum value in a given vector.

This is a static model which has also been used in [Thiele et al., 2007]. It disregards context switching times and it is valid for acyclic task graphs. We also assume that the application is computation-dominated (rather than communication-dominated), that is, the application throughput is not limited by the link bandwidths but only by the computation on the cores. As argued by Thiele et al. [2007], this model has a reasonable accuracy for typical streaming applications. Moreover, this static model holds for the simple data-driven scheduling of KPNs, thus it sets us free from the scheduling problem.

The computation objective is the minimization of the total execution time (equation 4.19) subject to capability constraints (equation 4.18). The objective function in equation 4.19 is not linear due to the max function. However, there is a linearization technique that transforms this equation to its linear counterpart by introducing new variables.

Linearization of $max()$ can be done by introducing a new variable:

$$\text{Minimize } max(x, y, z) \Rightarrow \text{Minimize } t \ \text{ subject to } \ t \geq x,\ t \geq y,\ t \geq z \tag{4.20}$$

4.2.3 Multi-objective optimization with ILP

We have defined two ILP problems that optimize two objectives separately. What we actually need is the multi-objective optimization of the combined problem that should result in a Pareto curve representing optimal solutions with different trade-offs for the two objectives. This is done by employing the ε-constraint

method [Chankong and Haimes, 1983]. This method relies on adding one of the objectives as a constraint by requiring it to be smaller than a chosen threshold. By solving the ILP problem several times for different values of the threshold and for a single objective, we obtain a Pareto curve. It is worth noting that the solutions found by the multi-objective ILP optimization are absolute optima unlike what would be obtained by employing evolutionary algorithms.

4.3 OTR: Online task remapping

Our overall goal is to enable the execution of KPN applications on NoC platforms in a fault tolerant manner. Chapter 5 describes in detail our approach to fault tolerance, specifically the recovery phase. The recovery phase enables isolation of the fault and continuity of operation, possibly with a degraded performance. The problem we are considering in the present chapter comprises one aspect of the recovery problem, that is, the algorithm that decides on the new task assignment configuration. This has to be a very fast algorithm, in order not to disrupt operation for long. We are mainly concerned with minimizing performance degradation. While achieving that we also aim at developing a solution that results in a short recovery time. The task remapping algorithm can work in two ways: *limited task migration* where only the tasks on the faulty core are migrated to other cores; and *unlimited task migration* where any task, even those on fault-free cores, can be migrated. The former will have a shorter reconfiguration time due to less number of tasks being migrated. However it will most likely result in a more degraded performance. The latter will certainly require a longer reconfiguration time while having a higher chance of less (or, ideally, non-) degraded performance.

We propose an optimal solution to the online task remapping problem based on our ILP formulation for both limited and unlimited task migration cases, and then present five different heuristics for the limited task migration case. Obviously the ILP solution cannot be applied at run-time. However, it makes it possible to measure the quality of the heuristic methods.

4.3.1 Optimal task remapping

In the case of unlimited task migration, we are able to obtain the Pareto curves for all single fault scenarios by adding the *faulty core* constraint given here below to the original ILP formulation.

Constraint *(faulty core)*: Given a faulty node n_f, a new constraint is added to the ILP formulation that forbids mapping of tasks on the faulty node n_f.

$$\sum_{j=1}^{|V_t|} X_{fj}^{NT} = 0 \tag{4.21}$$

In the case of limited task migration, the below constraint should be added as well.

Constraint *(migrate only tasks on the faulty core)*: given a faulty node n_f and an initial task mapping M^{NT}, a new constraint is added to limit the reconfiguration just to the tasks that are running on the faulty node n_f.

$$X_{ij}^{NT} = M_{ij}^{NT},\ 1 \leq i \leq |V_a|,\ 1 \leq j \leq |V_t|,\ i \neq f \tag{4.22}$$

For a totally heterogeneous NoC with all IP cores being different from each other, there can be $|V_a|$ successive single faults, eventually leading to no remaining fault-free cores. In the case of unlimited task migration, the total number of different architectural configurations N_F, from all fault-free cores to one fault-free core is

$$N_F = \sum_{i=1}^{|V_a|} \binom{|V_a|}{i} - 1 = 2^{|V_a|} - 2 \tag{4.23}$$

It means that we will need to calculate N_F Pareto curves for all these different scenarios. It is a heavy task even if it is done offline. One way of implementing the task remapping algorithm is by means of a look-up table [Lee et al., 2010] where we keep the resulting optimal mappings for all Pareto curves of the N_F different configurations. Assuming we have p points in a Pareto curve, encoding such information would cost B bits calculated as

$$B = (2^{|V_a|} - 2)\, p\, |V_t|\, \lceil log(|V_a|) \rceil \tag{4.24}$$

For a case with $|V_a| = 9$, $|V_t| = 12$, $p = 5$, we have $B = 14.94$ Kbytes. As the number of cores of same type increases and also depending on their placements in the NoC, this number decreases due to the occurrences of symmetrical configurations. In case the local memory in the tiles is restricted and/or the size of the NoC and the problem increases, this memory requirement may make it prohibitive to apply the look-up table technique.

4.3.2 Center of Gravity heuristic (CoG)

This heuristic places the task to be migrated in a core that resides in between the other tasks it communicates with by considering the amount of communication.

This heuristic takes into account only communication cost. More formally, let L_j be the set of tasks assigned to core n_j and let L_f be the set of tasks that reside on the faulty core n_f. Let $peers$ be a function that returns the list of tasks that a given task communicates with. Let $demand$ be a function that returns the bandwidth demand between two given tasks. Let $dist$ be a function that returns the Manhattan distance between two given nodes. The capability $M^{TN}_{cap\,ij}$ of node n_j being able to execute task t_i can be calculated as

$$M^{TN}_{cap} = M^{TC}_{cap} M^{CN} \tag{4.25}$$

The peer tasks of t_i that do not reside on the faulty node, $ext\_peers(t_i)$, are defined as

$$ext\_peers(t_i) = \{t_j \in peers(t_i) : \beta(t_j) \neq n_f\}. \tag{4.26}$$

Algorithm 1 CoG Algorithm

Require: initial mapping L, faulty node n_f
Ensure: new mapping L
1: **for all** $t_i \in L_f$ **do**
2: find a task $t_i \in L_f$ such that $\sum_{t_j \in ext\_peers(t_i)} demand(t_i, t_j)$ is maximum.
3: find $n_k \in V_a \wedge n_k \neq n_f \wedge M^{TN}_{cap\,ik} \neq 0$ such that $\sum_{t_j \in peers(t_i)} dist(n_k, \beta(t_j)) demand(t_i, t_j)$ is minimum.
4: $L_k \leftarrow L_k \cup \{t_i\}$, $L_f \leftarrow L_f \setminus \{t_i\}$
5: **end for**
6: **return** L

If there are tasks that communicate with each other and reside in the faulty node, the resulting mapping will depend on which order such tasks are being migrated. In algorithm 1, line 2, tasks are sorted with respect to their total bandwidth demands on their edges connected to the tasks on the non-faulty nodes and then migrated in descending order. In line 3, the algorithm selects the new node for t_i in such a way that the amount of communication due to t_i is minimized. The new node has to be of a core type that can realize the task t_i and it has to be a non-faulty core. In the event that there are still more than one candidate nodes equally satisfying the conditions, we choose the node with minimum computational load.

For the special case of mesh-based NoCs, the problem can be transformed to that of finding the center of gravity of masses by considering the communication demands as the masses of the external peer tasks. The center of gravity node can be found at once mathematically reducing the complexity from $O(n)$ to $O(1)$. Let

$coord$ be a function that returns the (x,y) coordinates of a given node in mesh-based NoC. The new node n_i for task $t_i \in L_f$ will have coordinates $coord(n_i)$

$$coord_i = \frac{\sum_{t_j \in peers(t_i)} coord(\beta(t_j))\, demand(t_j, t_i)}{\sum_{t_j \in peers(t_i)} demand(t_j, t_i)} \tag{4.27}$$

It is most likely that $coord_i$ will not have integer values, so we round it to obtain actual coordinates.

$$coord(n_i) = \lfloor coord_i + (0.5, 0.5) \rfloor \tag{4.28}$$

4.3.3 Nonidentical Multiprocessor Scheduling (NMS)

The objective regarding computation is equivalent to the scheduling of independent tasks on nonidentical processors in order to minimize the makespan (defined as the last finishing time of the given tasks). We adopt three heuristics (namely NMS-A, NMS-B and NMS-C) that have been proposed in [Ibarra and Kim, 1977] for this problem. They are slightly different from each other and it has been shown that there are examples in which each of them is superior to others. In terms of the number of processing nodes (m) and tasks on the faulty node (n), NMS-A/B/C have different orders of complexity, that is, $O(mn)$, $max(O(n\,log\,n), O(mn))$, $O(mn^2)$, respectively. When remapping the tasks, these heuristics take into account only the total execution time. It may be the case that the resulting remapping does not satisfy the capacity constraints (equation 4.11).

NMS-A: 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. NMS-A is given in algorithm 2. In line 2, the task $t_i \in L_f$ is remapped on the core that minimizes its finishing time.

NMS-B: For each task $t_i \in L_f$, NMS-B algorithm first orders the tasks in L_f according to decreasing $min\{T_{cap\,ij}^{TN} : 1 \leq j \leq |V_a|\}$, and then calls NMS-A.

NMS-C: This algorithm iteratively remaps the tasks by choosing a task from L_f that gives the least finishing time as shown in algorithm 3. The order of tasks to be remapped is not known a priori as in NMS-A or NMS-B. As shown in line 2, the task and its new node are searched simultaneously.

Algorithm 2 NMS-A Algorithm

Require: initial mapping L, n_f, T^N before fault, T^{TN}_{cap}
Ensure: new mapping L
1: **for all** $t_i \in L_f$ **do**
2: find the smallest j such that $T^N_j + T^{TN}_{cap\,ij} \leq T^N_l + T^{TN}_{cap\,il}$ for all $1 \leq l \leq |V_a|$, $l \neq f$
3: $L_j \leftarrow L_j \cup \{t_i\}$, $L_f \leftarrow L_f \setminus \{t_i\}$
4: $T^N_j \leftarrow T^N_j + T^{TN}_{cap\,ij}$
5: **end for**
6: **return** L

Algorithm 3 NMS-C Algorithm

Require: initial mapping L, n_f, T^N before fault, T^{TN}_{cap}
Ensure: new mapping L
1: **while** $L_f \neq \emptyset$ **do**
2: find a task $t_i \in L_f$ and a node $n_j \in V_a \wedge n_j \neq n_f$ such that $T^N_j + T^{TN}_{cap\,ij} \leq T^N_j + T^{TN}_{cap\,kj}$ for all $t_k \in L_f$ and $T^N_j + T^{TN}_{cap\,ij}$ is minimum.
3: $L_j \leftarrow L_j \cup \{t_i\}$, $L_f \leftarrow L_f \setminus \{t_i\}$
4: **end while**
5: **return** L

4.3.4 Localized NMS Heuristic (LNMS)

The heuristics proposed above take into account either communication or computation. In order to develop a heuristic that performs well for both objectives, we limit the region of nodes where we employ the NMS heuristics. This algorithm is called the Localized NMS (LNMS), and communication cost is bounded by selecting a remapping region for each task that falls in between the peer tasks.

We define a parametrized $region(t_i, s)$ function that takes an integer s and returns the set of nodes that have at most Manhattan distance s to the node at the center of gravity of the peer tasks of the given task t_i.

$$region(t_i, s) = \{n_j \in V_a : dist(n_j, CoG(L, n_f)_i) \leq s\} \tag{4.29}$$

For $s = 0$, LNMS reduces to CoG (i.e., $LNMS(0) \equiv CoG$) and for $s = s_{max}$ such that $region(t_i, s_{max}) = V_a$, it reduces to NMS (i.e., $LNMS(s_{max}) \equiv NMS$). Therefore we should be able to obtain a sub-optimal Pareto curve for the range $1 \leq s \leq s_{max}$ that represents different trade-off points between communication and computation.

All three NMS heuristics can have localized versions such as LNMS-A(s), LNMS-B(s) and LNMS-C(s). Rather than rewriting the whole pseudocode, we highlight the differences in each case:

- *LNMS-A(s) and LNMS-B(s)*: in line 2, instead of $1 \leq j \leq |V_a|$, we have $n_j \in \text{region}(t_i, s)$;
- *LNMS-C(s)*: in line 2, instead of $n_j \in V_a$, we have $n_j \in \text{region}(t_i, s)$.

4.4 The reliability aspect

This section focuses on the reliability aspect of the *fault-aware online task remapping* technique. We also look at *N-modular redundancy* [Derin, Diken and Fiorin, 2011], an alternative technique that makes use of concurrent self-checking by executing redundant application tasks. We consider the problem at the level of application abstracting from the underlying hardware solution. In our study, we adopt as a reliability metric the MTTF. For each technique, we propose an analytical model for calculating the MTTF of the overall system. We use the same notation presented in table 4.1.

4.4.1 Reliability estimation for online task remapping

Reliability when applying online task remapping is estimated by building a *fault tree* of the application and by calculating a corresponding value for the MTTF.

Creating the fault tree

In order to calculate the reliability of a system provided with online task remapping ability, we build a fault tree [Vesely et al., 1981] starting from the application and the platform specifications (see algorithm 4). Given information about the tasks that each core can execute, and the type of core in each node, the algorithm creates a fault tree based on the idea that the application will fail when there will be no fault-free cores left for any of the core types required by the application.

In fault trees, leaf nodes, intermediate nodes, and the root node represent basic events, complex events and the top event, respectively. Basic events are denoted by circles, and complex or top events by rectangles as shown in figure 4.1(b). In our algorithm, failed node events correspond to basic events (represented as n_i in algorithm 4). The algorithm creates complex events at two levels in a bottom up manner. First complex event type is the failed core type

Algorithm 4 create_fault_tree(M^{TC}_{cap}, M^{NC})

Require: M^{TC}_{cap}, M^{NC}
Ensure: E_{fault_tree}
1: BoolExp $E_{fault\_tree} = false$
2: **for all** $i \in [1..|V_t|]$ **do**
3: BoolExp $E_{t_i} = true$
4: **for all** $j \in [1..|C|]$ **do**
5: BoolExp $E_{C_j} = true$
6: **if** $M^{TC}_{cap_{ij}} = 1$ **then**
7: **for all** $k \in [1..|V_a|]$ **do**
8: **if** $M^{NC}_{kj} = 1$ **then**
9: $E_{C_j} = E_{C_j} \wedge n_k$
10: **end if**
11: **end for**
12: **end if**
13: $E_{t_i} = E_{t_i} \wedge E_{C_j}$
14: **end for**
15: $E_{fault\_tree} = E_{fault\_tree} \vee E_{t_i}$
16: **end for**
17: **return** E_{fault_tree}

event (E_{C_j} in line 5). A conjunctive set of failed nodes belonging to the same core type leads to a failed core type event (line 9). Second type of complex event is the failed task event (E_{t_i} in line 3). A task fails when no core type is left that can execute that task, thus a failed task event occurs when that conjunctive set of failed core type events occurs (line 13). Finally, the algorithm creates the top event, E_{fault_tree}, representing the failure of the application. The application fails when any task fails, as denoted with the disjunctive set of all failed task events (line 15). Figure 4.1(b) shows the fault tree that results from the case of an application with four tasks running with online task remapping fault tolerance technique on a 2×2 NoC platform shown in figure 4.1(a). The core types (C_i) capable of running each task (t_j) are presented as a table in the figure.

Calculating MTTF from the fault tree

As a second stage, we calculate the MTTF of the application on the given platform starting from the created fault tree. In doing that, we adopt the binary decision

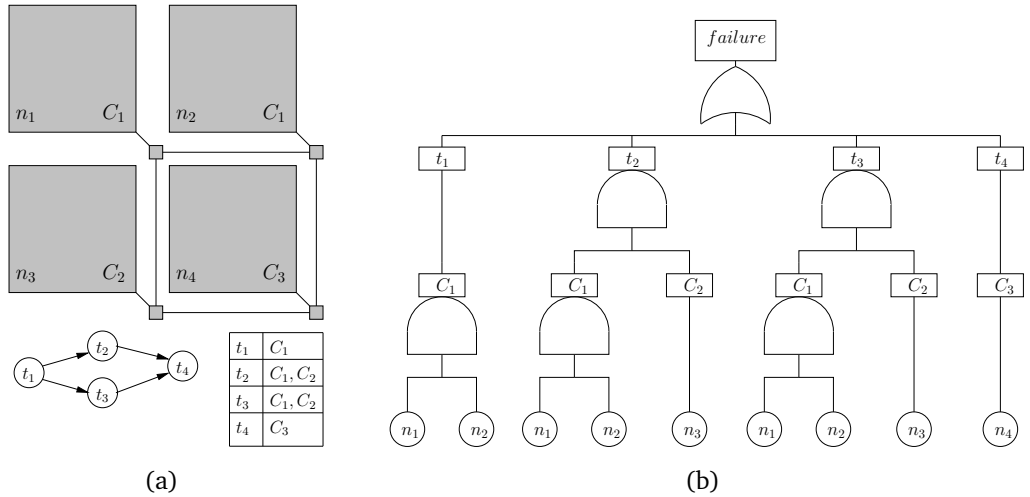


Figure 4.1. A 2×2 NoC, a simple task graph and a table listing tasks and the core types capable of executing the tasks (a), and the corresponding fault tree (b)

diagram (BDD) based approach [Sinnamon, 1996] which does not suffer from the approximation errors due to the multiple occurring events (MOEs) in the fault tree when it is evaluated with the Kinetic Tree Theory [Vesely, 1971]. Given the procedure in algorithm 4, the resulting fault tree is prone to having MOEs when there are core types that can execute different tasks or when there are more than one instances of a task. In the BDD-based approach, the fault tree, which is actually a boolean expression, is captured as a BDD. The set of all the paths leading to 1s are called satisfying paths, Sat. A satisfying path assigns values to nodes as 1 (n_i, failure) and 0 ($\bar{n}_i$, non-failure). The probabilities of satisfying assignments are summed in order to calculate the overall probability of failure ($Q_{sys}(t)$). Given the probability that processing node n_i will fail at time t, $Pn_i(t)$, the path probabilities are calculated by multiplying the failure ($Pn_i(t)$) or non-failure probabilities ($1-Pn_i(t)$) of the BDD nodes depending on their assignments.

$$Q_{sys}(t) = \sum_{s_i \in Sat} (\prod_{n_j \in s_i} Pn_j(t) \prod_{\bar{n}_k \in s_i} (1 - Pn_k(t))) \tag{4.30}$$

$$MTTF_{sys} = \int_0^\infty R_{sys}(t)dt \tag{4.31}$$

where reliability of the system, $R_{sys}(t) = 1 - Q_{sys}(t)$.

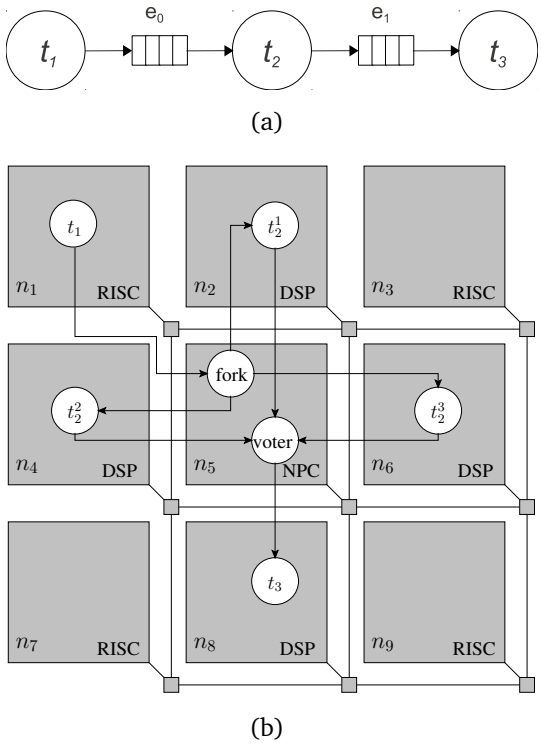
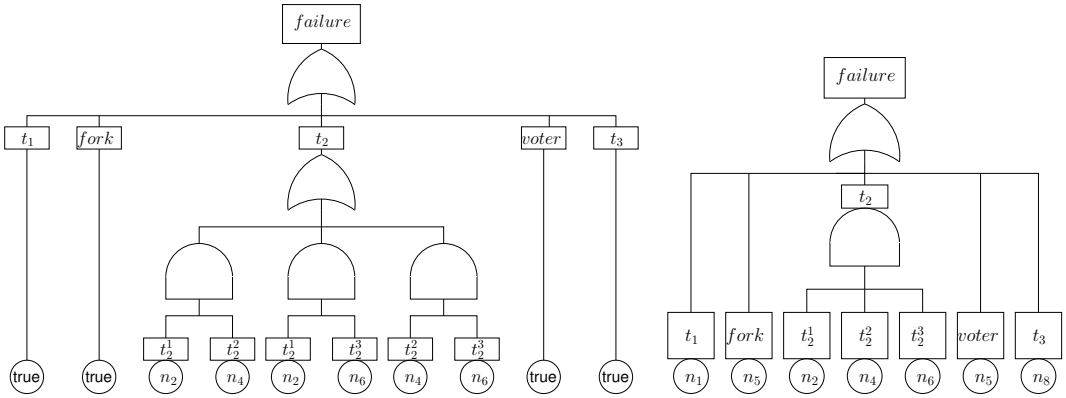


Figure 4.2. Example of a KPN application composed of three software tasks (a), and a mapping of the example application using the TMR pattern onto a 3 × 3 NoC (b)

As a result, we are able to calculate the MTTF of an application on a given platform that supports online task remapping. This can help us in designing a platform for our application in a reliability-aware manner by properly selecting core types (M_{cap}^{TC}) and core mapping (M^{NC}).

4.4.2 Reliability estimation for N-modular redundancy

N-modular redundancy (NMR) is based on the idea of transforming the original task graph by replicating some of the tasks N times. Outputs of the replicated tasks are collected and given to a majority voter, which decides on the final result according to the most recurrent value at its input. Unlike online task remapping, the NMR technique described below allows various reliability levels, even if the platform is fixed, by applying some patterns that introduce redundancy at the application level and by exploiting different mappings of tasks on the platform [Derin, Diken and Fiorin, 2011].



(a) The case of safe failure (b) The case of unsafe failure

Figure 4.3. The fault trees corresponding to the mapping in figure 4.2(b)

In the case of a single task, parallel instances of the task are created on different cores along with fork tasks for every input channel, and majority voter tasks for every output channel. Figure 4.2(b) exemplifies the transformation of the task graph in figure 4.2(a) with the NMR pattern. A fork task creates a copy of the incoming message for each redundant instance and forwards each copy to the input channels of those instances. A majority voter task reads a token from all of its input channels and finds out the most recurrent token and sends it to its output channel. If a different input token is detected, the voter can signal that the core producing that token is faulty.

There are various ways to apply the NMR pattern to electronic circuits and systems [Koren and Krishna, 2007]. In the case of task graphs, one strategy, that we call *NMR-per-single-task*, is to encapsulate single tasks within fork and voter tasks. The other strategy, that can be named *NMR-per-multiple-tasks*, is to encapsulate multiple tasks within fork and voter tasks. In the former case, the number of forks and voters will be greater than the latter case, thus leading to more overhead in communication cost. However, the former case is expected to yield better reliability. In this work, we investigate therefore the first strategy and provide algorithms to calculate the reliability metric of applications for which NMR is applied distinctly on each task.

Creating the fault tree

In order to calculate the reliability of a KPN application that is transformed with the NMR pattern and mapped onto a NoC platform, we derive a fault tree by analyzing all fault cases that lead to failure. In order to make our approach clear,

Algorithm 5 create_fault_tree algorithm (safe failure - inability to provide checked results)

1: BoolExp $E_{fault\_tree} = false$
2: **for all** $t_i \in V_t$ **do**
3: BoolExp $E_{t_i} = false$
4: **if** $transform(t_i) = t_i$ **then**
5: $E_{t_i} = true$
6: **else**
7: **for all** $t_i^j \in transform(t_i)$ **do**
8: BoolExp $E_{con\_i} = true$
9: **for all** $t_i^k \in transform(t_i)$ **do**
10: **if** $t_i^k \neq t_i^j$ **then**
11: $E_{con\_i} = E_{con\_i} \wedge \beta(t_i^k)$
12: **end if**
13: **end for**
14: $E_{t_i} = E_{t_i} \vee E_{con\_i}$
15: **end for**
16: **end if**
17: $E_{fault\_tree} = E_{fault\_tree} \vee E_{t_i}$
18: **end for**
19: **return** E_{fault_tree}

we consider an example application in which three tasks t_1, t_2 and t_3 are connected as a pipeline. The TMR pattern is applied to t_2 leading to three instances, namely t_2^1, t_2^2 and t_2^3 along with a *fork* and a majority *voter*. As shown in figure 4.2(b), this application is mapped onto a 3×3 NoC-based platform with a specialized node n_5 that hosts the *fork* and *voter* tasks implemented in hardware as non-programmable cores.

In the case of an NMR-ed application, there can be two types of failure: *safe failure* and *unsafe failure*. A safe failure is the failure of the system to provide *checked* results. Unsafe failure is the failure of the system to provide *correct* results. A safe failure happens if there is only one instance left of any task type. The example in figure 4.2(b) is already in failing mode if evaluated from the point of view of safe failures because t_1 and t_3 have single instances and their results are not checked. Figure 4.3(a) shows the derived fault tree that results from the mapping shown in figure 4.2(b) when considering safe failure.

Let $g_t^R = (V_t^R, E_t^R)$ be the task graph that results from the the application of self-checking patterns to g_t. Let $transform(t) : V_t \rightarrow V_t^R$ be a function that

returns the set of redundant instances in g_t^R of task t. The procedure to obtain the fault tree related to the failure of providing checked results is listed in algorithm 5.

Basic events (n_i inside a circle in figure 4.3(a)) correspond to the failure of the processing node n_i. The algorithm creates complex events at two levels in a bottom up manner. A complex event at the first level represents a failed redundant task instance due to the failure of its processing node.

The second level represents task type failures (E_{t_i}, in line 3). If there exists at least one task without redundant instances, it leads to a fault tree that always evaluates to true (line 5). For a task with redundant instances, the failure of the task type is implied when there remains no longer a pair of redundant instances running on two different fault-free nodes (lines 6–16).

Finally, the top event represents the failure of the application when any one of the task types fails (line 17). Due to the assumption of highly fault-tolerant interconnection network, the failure characteristics of the overall application can be obtained by evaluating the derived fault tree given only the failure characteristics of the processor cores.

In the case of unsafe failure, the procedure to obtain the fault tree is listed in algorithm 6. For non-redundant tasks, task type failure occurs if the processor that hosts the task fails (line 5). For redundant tasks, the failure of the task type is obtained by conjugating failed redundant task instance events (lines 6–10). The top event (i.e., the failure of the application) occurs if there is no instance left of any task type (line 11). Figure 4.3(b) shows the derived fault tree that results from the mapping shown in figure 4.2(b) when considering safe failure. The application will fail if either of n_1, n_5, n_8 fails or if all of n_2, n_4 and n_6 fail.

In order to evaluate the MTTF of the fault tree, we adopt again the BDD-based approach described in section 4.4.1. For the example in figure 4.3(b), the set of all satisfying assignments is $Sat = \{\bar{n}_1\bar{n}_2\bar{n}_5n_8,\ \bar{n}_1\bar{n}_2n_5,\ \bar{n}_1n_2\bar{n}_4\bar{n}_5n_8,\ \bar{n}_1n_2\bar{n}_4n_5,\ \bar{n}_1n_2n_4\bar{n}_5\bar{n}_6n_8,\ \bar{n}_1n_2n_4\bar{n}_5n_6,\ \bar{n}_1n_2n_4n_5,\ n_1\}$.

Assuming a Poisson distribution of faults with hazard rate λ_i for n_i, we have the probability that n_i fails at time t as $Pn_i(t) = 1 - e^{-\lambda_i t}$. In consequence, the probability that n_i does not fail at time t is given by $P\bar{n}_i(t) = e^{-\lambda_i t}$. In the example, assuming $\lambda = 10^{-5}$ for the RISC and DSP processors and $\lambda_5 = 10^{-7}$ for n_5 that hosts the fork-voter core, we obtain $MTTF_{sys} \approx 10^3\ time\ units$.

Algorithm 6 create_fault_tree algorithm (unsafe failure - inability to provide correct results)

1: BoolExp $E_{fault\_tree} = false$
2: **for all** $t_i \in V_t$ **do**
3: BoolExp $E_{t_i} = true$
4: **if** $t_i = transform(t_i)$ **then**
5: $E_{t_i} = \beta(t_i)$
6: **else**
7: **for all** $t_i^j \in transform(t_i)$ **do**
8: $E_{t_i} = E_{t_i} \wedge \beta(t_i^j)$
9: **end for**
10: **end if**
11: $E_{fault\_tree} = E_{fault\_tree} \vee E_{t_i}$
12: **end for**
13: **return** E_{fault_tree}

4.5 Experimental results

For evaluating our proposals, we developed the *maponoc* tool[1]. It consists of mainly two programs. The *ILP-based mapper* program solves the optimal remapping problems by using the API of the IBM ILOG CPLEX optimizer [*IBM ILOG CPLEX Optimizer*, n.d.]. Its flow is depicted in figure 4.4(a). Given input files that describe the application (the task graph along with bandwidth demands of data dependencies), the platform (architecture graph, core types, core mapping, routing algorithm), offline profiling results (task workloads on the available core types) and the fault scenario (the initial mapping and the set of faulty nodes), the tool provides as output the Pareto-optimal remappings. It can also be used for the mapping problem (i.e., without providing the fault scenario). It can operate in time-limited, gap-limited or unlimited modes. In the first mode, an upper bound on the time to run each optimization is set by the user. In the second mode, the optimizer returns the current best solution when it reaches a certain proximity to the optimal solution (given as a percentage gap). In the third mode, the execution of the tool is not limited by a time or a gap value. For all the results reported in this section, we have used the unlimited mode.

The reliability metric, which was introduced in section 4.4 as a third objective to the mapping problem, is not linear. Therefore we implemented the *genetic*

[1]Available as open-source in `https://github.com/derino/maponoc`

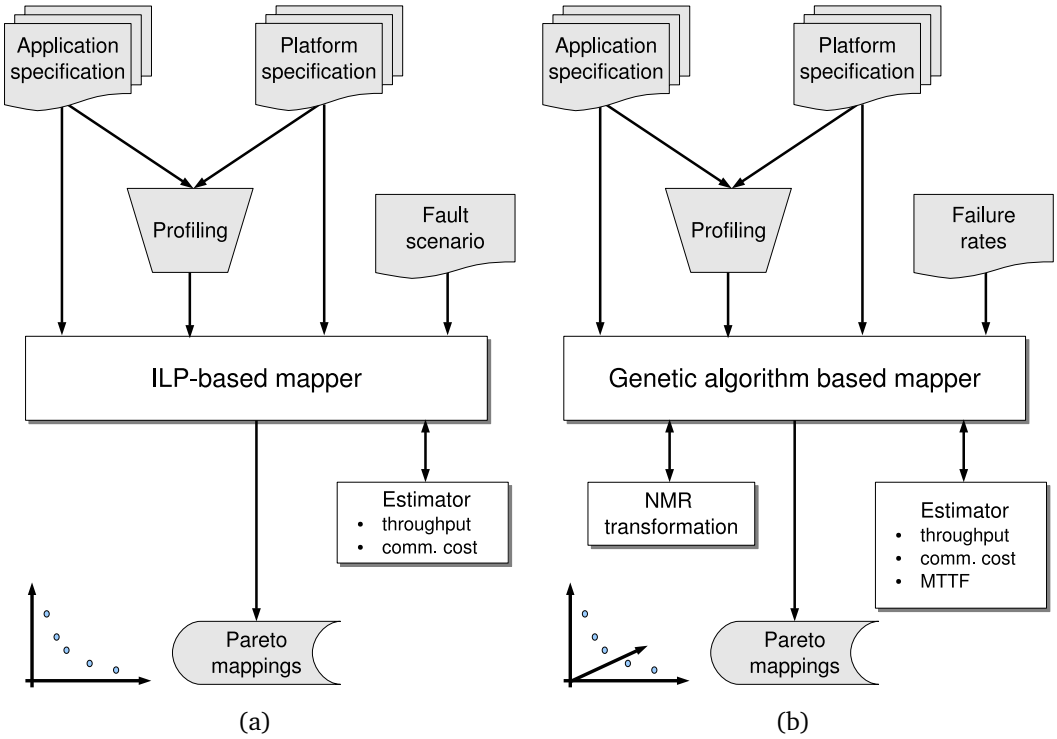


Figure 4.4. Tool flow for evaluating the fault-aware online task remapping with the ILP-based mapper (a) and the reliability estimation with the genetic algorithm based mapper (b)

algorithm based mapper program (based on NSGAII with constraints [Deb et al., 2002]) that calculates the Pareto mappings for the three objectives: throughput, communication cost and MTTF subject to the link bandwidth capacity constraints and core capability constraints. Its flow, given in figure 4.4(b), is similar to the one of the ILP-based mapper except that it requires the failure rates of each core as input rather than the fault scenario. In addition, it incorporates an MTTF estimator and a block that transforms the application using the NMR pattern. In the genetic algorithm, a mapping represents an individual. The mutation operation is defined by swapping the mapping of two randomly selected tasks, while the cross-over operation merges the mapping of a random task set from one parent with the complementary set from the other parent. The program also accepts the size of the population (P), number of generations (G), mutation (M) and cross-over (X) rates as input from the user. For the results reported in this section, we have used the following parameter values: $(P, G, M, X) = (1000, 100, 0.001, 0.8)$.

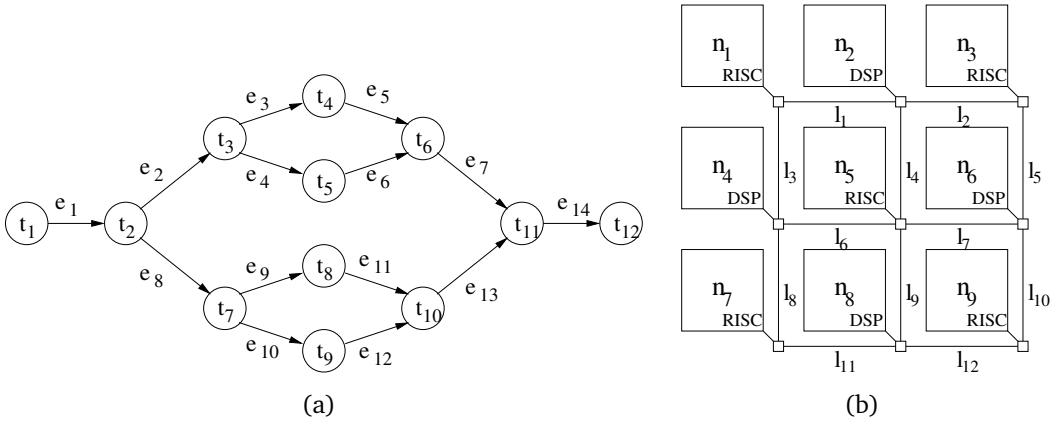


Figure 4.5. An MPEG-2 encoder task graph with 12 tasks (a), and a 3 × 3 mesh-based NoC with RISC and DSP processors (b)

4.5.1 Case study: the MPEG-2 decoder

To illustrate the problem formulation better, firstly we use as a running example the widely used MPEG-2 decoder benchmark [Thiele et al., 2007], whose task graph is shown in figure 4.5(a), and the XY-routing based NoC architecture shown in figure 4.5(b). The throughput of the links of the NoC is 100 MBps. The test video is 15 seconds long with a resolution of 704 × 576 pixels and the frame rate is of 25 frames per second (fps).

M^{TE} can be obtained from the task graph. Deriving M^{NP} and M^{PL} is trivial given the architecture and the routing algorithm. The offline profiling data (T^{CT}_{cap}) and bandwidth demands of the data dependencies (d) are given in tables 4.2 and 4.3 respectively. Other parameters are listed as follows:

$$c_i = 100 \text{ MBps}, 1 \leq i \leq |E_a|$$

$$C = \{C_1, C_2\} = \{\text{RISC}, \text{DSP}\}$$

$$M^{TC}_{cap} = \mathbf{1}_{12\times 2}$$

The MPEG-2 case study has been solved by using our ILP-based mapper. Figure 4.6(a) shows the Pareto curve obtained when all nodes are working. Considering that it was a 15 seconds long video clip, the solutions that satisfy the frame rate requirement are those that have a total execution time of less than 15 seconds. One can choose the mapping among them by trading off the total execution time and the communication load.

Table 4.2. Execution times (in seconds) of tasks on the available core types (T_{cap}^{CT})

Core type	Tasks											
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
RISC	0.13	6.68	0.06	2.00	2.00	0.05	0.06	2.00	2.00	0.05	12.33	0.18
DSP	0.20	8.52	0.04	1.25	1.25	0.04	0.04	1.25	1.25	0.04	8.51	0.30

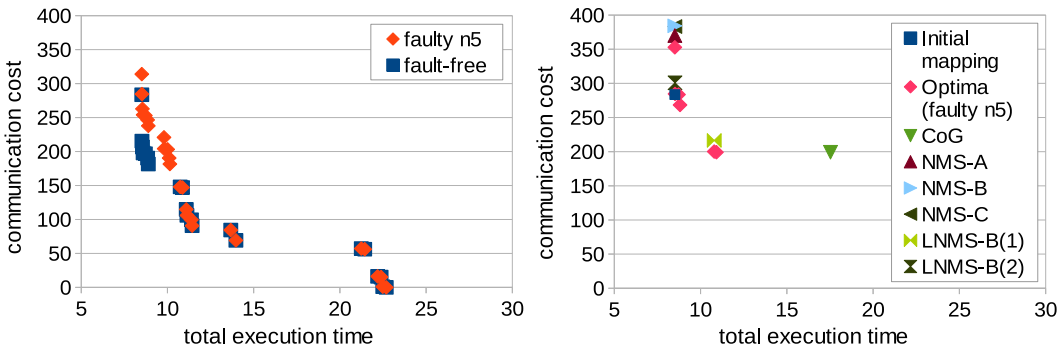
Table 4.3. Bandwidth demands (in MBps) of edges (d)

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}
1.0	34.6	28.1	28.1	28.1	28.1	65.0	34.6	28.1	28.1	28.1	28.1	65.0	15.2

Figure 4.6(a) also shows the Pareto-optimal remappings in the case of faulty n_5 and unlimited task migration. We see that in the regions where the execution time constraint is relaxed (e.g., execution time $> 10s$), the Pareto remappings coincide with the Pareto points found for the case in which all the cores are fault-free. The reason is that, for those Pareto points, tasks are mostly mapped onto few nodes. Therefore, the absence of n_5 is not relevant. But for the initial Pareto points where the execution time is smaller than $10s$, the degradation is visible, meaning that the Pareto remappings do not coincide with the initial Pareto points. This is due to the fact that for keeping the same execution time ($8.51s$), the required parallelism is high, and thus almost every task should be mapped onto a different node. In this case, the absence of n_5 becomes critical. Moreover, since n_5 is the central node, the communication cost of alternative mappings increases.

The limited task migration case has also been considered. Starting with an initial optimal mapping ($t_7, t_8, t_9, t_{10} \rightarrow n_1; t_{11} \rightarrow n_2; t_3, t_4, t_5, t_6 \rightarrow n_3; t_1, t_2, t_{12} \rightarrow n_5$), the Pareto-optimal remappings have been calculated for the faulty n_5 case. Table 4.4 lists the Pareto-optimal remappings in the limited case along with the performance degradation ratios with respect to the initial mapping situation prior to the fault.

We have also calculated the remappings for the faulty n_5 scenario by using the CoG, NMS-A/B/C and LNMS-A/B/C(s) heuristics for $0 < s < s_{max}$. Table 4.5 lists these results providing also the degradation with respect to the initial mapping. A subset of the results for the case of limited task remapping is also presented in figure 4.6(b). It shows the proximity of the remapping results of 7 heuristics to the metrics of the initial mapping and the Pareto-optimal remappings. The results reveal that for each heuristic (except CoG) there is a remapping point close to each optimal remapping point, thus showing that the proposed heuristics cover various trade-offs among the objectives.



(a) Pareto-optimal mappings and remappings (unlimited task migration)

(b) Comparison of results between heuristics, optimal remappings and the initial mapping (limited task migration)

Figure 4.6. Remapping results for the MPEG2 decoder case study on a 3×3 heterogeneous platform with a faulty node (n_5)

Table 4.4. Degradation achieved by Pareto-optimal limited remappings for faulty n_5 scenario

faulty node	**mapping**	**exe.time (obj. 1)**	**com.cost (obj. 2)**	**degradation (%) obj. 1**	**obj. 2**
none	initial	8.51	283.6		
n_5	Pareto1	8.51	352.8	0	24
	Pareto2	8.52	284.6	0	0
	Pareto3	8.72	283.6	2	0
	Pareto4	8.81	268.4	4	-5
	Pareto5	10.79	200.2	27	-29
	Pareto6	10.92	199.2	28	-30

4.5.2 A synthetic task graph

To evaluate the heuristics further, we created a synthetic task graph with 30 tasks and considered a 4×4 mesh NoC composed of homogeneous cores. By varying the throughput constraint of the application we obtained 13 Pareto optimal initial mappings. For all the 13 initial mappings and all 16 single fault scenarios, we obtained the Pareto optimal remappings as well as the remappings calculated by the heuristics. We report the degradations incurred with respect to the initial mapping in figure 4.7(a), averaged over all the initial mappings and all the fault scenarios. By considering the execution time objective, it can be seen that the average degradation keeps decreasing from one extreme (CoG) to another ex-

Table 4.5. Degradation achieved by proposed heuristics for faulty n_5 scenario

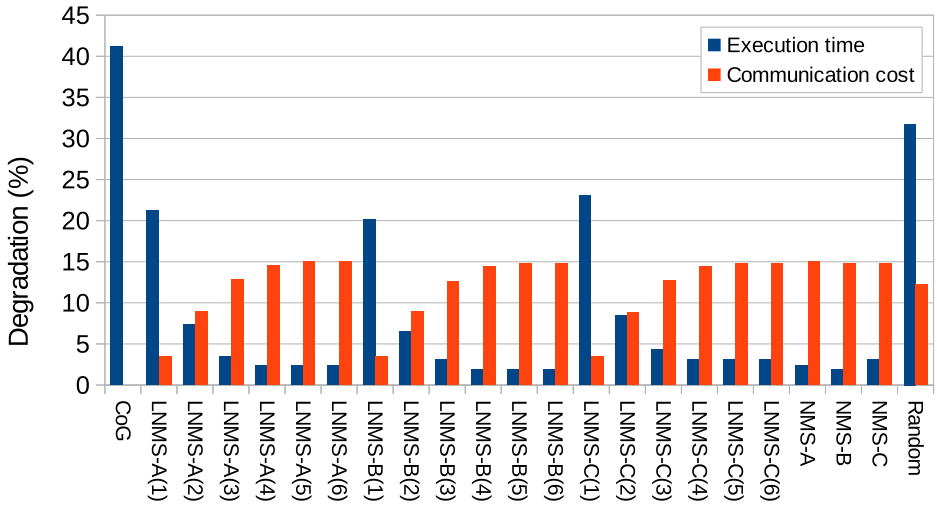
faulty node	method	exe.time (obj. 1)	com.cost (obj. 2)	degradation (%) obj. 1	obj. 2
none	initial	8.51	283.6		
n_5	CoG	17.53	199.2	105.99	-29.76
	NMSA	8.51	370	0	30.47
	NMSB	8.51	384.2	0	35.47
	NMSC	8.51	383.2	0	35.12
	LNMS-A(1)	10.79	216.4	26.79	-23.7
	LNMS-B(1)	10.79	216.4	26.79	-23.7
	LNMS-C(1)	10.92	214.4	28.32	-24.4
	LNMS-A(2)	8.52	300.8	0.12	6.06
	LNMS-B(2)	8.52	300.8	0.12	6.06
	LNMS-C(2)	8.52	370	0.12	30.47

treme (NMS), and vice versa for the communication cost objective. The results obtained for the LNMS heuristics constitute the intermediate values, with the s parameter ranging from 1 to 6. This confirms that the LNMS presents a trade-off opportunity between the two objectives. For $s = 2$, the average degradations for both objectives fall below 9%. As a conclusion, when applying these techniques in practice, the s parameter can be searched at run-time. By starting with the middle value of its range and checking the heuristic results against the constraints of the system (throughput and communication cost in this case), or the allowed degradation levels, the search algorithm can try to increase or decrease the values of s until the best acceptable result is achieved.

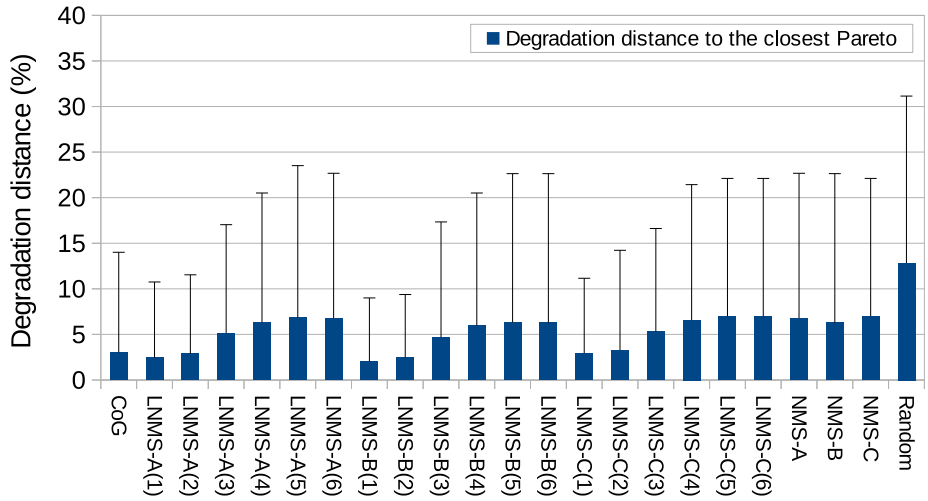
The quality of the heuristics can be assessed by evaluating the proximity of their results to the optimal remapping decisions. In order to evaluate the distance from the optimal remapping, we define a *degradation distance* metric. For an initial mapping point with objective values i_1 and i_2, $L_i = (i_1, i_2)$, a Pareto-optimal remapping point $L_p = (p_1, p_2)$ and a heuristic remapping point $L_h = (h_1, h_2)$, we calculate the degradation distance δ as

$$\delta = \sqrt{(\frac{h_1 - p_1}{i_1})^2 + (\frac{h_2 - p_2}{i_2})^2} \tag{4.32}$$

When calculating the degradation of a remapping point obtained by a heuristics with respect to the Pareto-optimal remapping curve, we consider the Pareto point that has the smallest degradation distance to the heuristic point. This is a logical decision because it makes sense to compare solutions that adopt a similar trade-



(a) Degradation achieved by proposed heuristics with respect to the initial mapping



(b) Additional degradation incurred by proposed heuristics with respect to the closest Pareto-optimal remapping (mean and 95 percentile are shown)

Figure 4.7. Remapping results for 30 tasks on a 4×4 homogeneous platform averaged over 13 initial mappings and 16 single fault scenarios

off among the objectives.

We report the degradation distances of the heuristics with respect to the optimal remapping solutions obtained by the ILP-based mapper in figure 4.7(b). The figure displays the average degradation and the 95 percentile evaluated over all

13 initial mappings and 16 fault scenarios. The figure shows that for all heuristics the average degradation distance is smaller than 7%, and that in 95% of the cases the degradation distance remains below 23%.

In addition to our heuristics, we implemented a routine that remaps tasks randomly in order to assess whether the time spent to execute the heuristics pays off and leads to better results than a purely random remapping or not. We executed random remapping 10 times per fault scenario. In average, random mapping leads to significantly worse results in terms of execution time compared to the heuristics (32% degradation with respect to the initial mapping). In terms of communication costs, it is comparable to LNMS($s_{max}/2$). This is simply because, in average, the *random* heuristic maps tasks within the region with radius $s = s_{max}/2$ (i.e., the expected value of a uniformly random number in the range $[1, s_{max}]$).

The total number of binary decision variables in the mapping problem are $|V_t| \times |V_a| + |E_t| \times 2 \binom{|V_a|}{2}$. In order to assess scalability of the ILP method, we solved the optimal mapping problem of 2 and 3 MPEG-2 decoders onto a 3×3 mesh NoC, which produced two more task graphs with 24 and 36 tasks. The time for the ILP-based mapper to obtain these solutions was in average 0.78s, 27.87s and 29 min per Pareto point for the cases with 12, 24 and 36 tasks, respectively. With regard to the NoC size, for the two cases of a 4×4 NoC with 30 tasks and of a 6×6 NoC with 70 tasks, the average solution time of the Pareto remapping points per fault scenario was 75s and 82.5 min, respectively. As mentioned earlier, the ILP-based mapper can be operated in a time-limited or gap-limited manner if the solution time becomes prohibitively long.

In order to assess the computation time of the remapping, that is, the execution times of the heuristics, we executed them on one typical embedded processor, i.e., a Xilinx Microblaze processors [Xilinx, 2010] with 64 KB of instruction and data cache, running at 100 MHz, included as a core of a NoC platform. We considered the case of a 4×4 mesh NoC with 30 tasks mapped onto its cores, and evaluated the time needed for the execution of each heuristic for all the 16 single fault scenarios.

Figure 4.8 shows, in Millions of cycles (Mcc), the average time employed by each heuristic. The first thing to notice from the results is that computation time changes almost linearly with respect to the s parameter. Considering that LNMS is composed of a call to CoG, and to the *region()* and NMS functions, the increase in time can be accounted for by the increase in the computation time of the *region()* and NMS functions given by the increase in the value of s. Secondly, the results obtained for the NMS computation times are a slightly different than the theoretical expectations. NMS-B takes longer than NMS-C. This is due to the

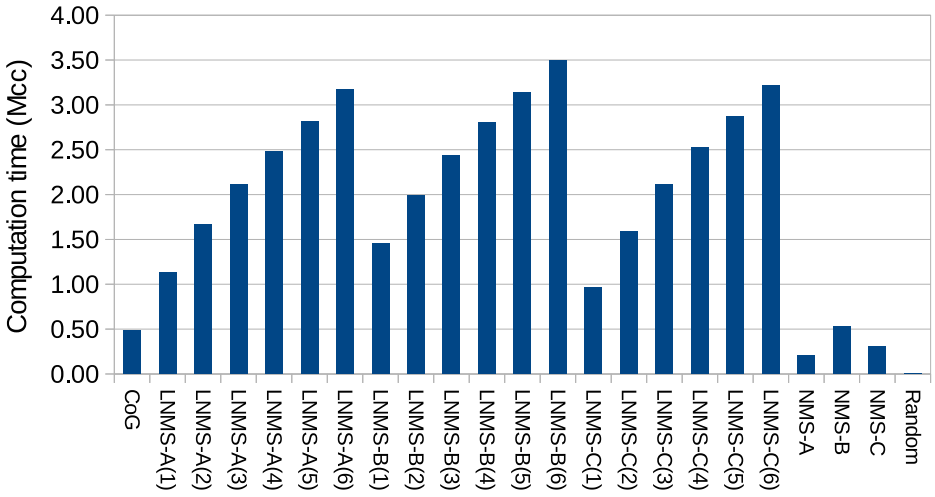


Figure 4.8. Computation times of the heuristics on the Microblaze processor

additional overhead of calling the qsort() function in the C standard library (used when implementing NMS-B) which becomes more noticeable on the considered use case that has a small average number of tasks on the faulty node.

4.5.3 Case studies on the platform

We chose as case studies two streaming applications in the multimedia domain, i.e., an M-JPEG encoder and a H.264 decoder. We run them on the 2×2 mesh of general-purpose processors, as detailed in section 3.1, implemented on a Virtex-6 FPGA board. The two applications are described below.

M-JPEG encoder

The PPN specification of the M-JPEG encoder is shown in figure 4.9. 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. Figure 4.9 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 4.6. Comparing these numbers with the amount of inter-process communication it can be inferred that this application has a high computation/communication ratio.

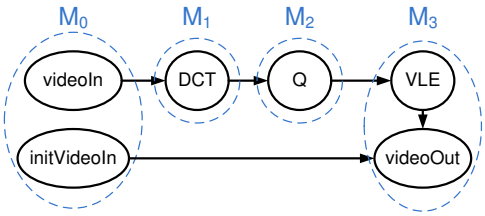


Figure 4.9. PPN specification of the M-JPEG encoder.

Table 4.6. Execution times of M-JPEG processes

Process	Avg. execution time (c.c.)
M_0	1923
M_1	123626
M_2	69254
M_3	47989

H.264 decoder

The simplified PPN specification of this case study is shown in figure 4.10. In the final implementation, the tasks 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 4.7.

4.5.4 Evaluation of the remapping strategies on the platform

In this section, the quality of the heuristics is assessed through M-JPEG and H.264 case studies by evaluating the accuracy of the remapping evaluations obtained by the NMS-A/B/C heuristics with respect to the actual measurements taken on a real implementation.

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 figure 4.11(a), we consider two single fault scenarios for n_1 and n_2. As shown in figure 4.11(b), 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, figure 4.11(c) 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

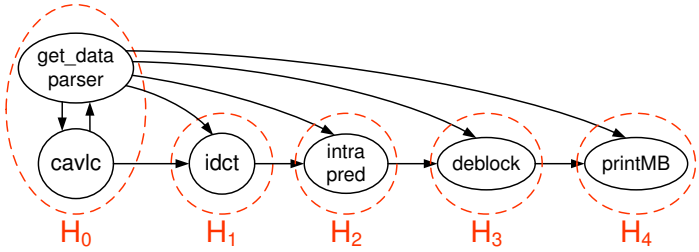


Figure 4.10. Simplified PPN specification of the H.264 decoder.

Table 4.7. Execution times of H.264 processes

Process	Avg. execution time (c.c.)
H_0	95643
H_1	55775
H_2	33645
H_3	9724
H_4	4075

total execution times of the M-JPEG application for all possible remappings, T_{R_i}, are measured on the platform 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 equation 4.33.

$$\text{Performance degradation}(R_i) = \frac{T_{R_i} - T_I}{T_I} \tag{4.33}$$

Measured and calculated values are used in equation 4.33 for calculating the *measured* and *analytical model* degradation results shown in figures 4.12(a) and 4.12(b) for faulty n_1 and faulty n_2 cases, respectively. Note that in some cases, for instance R_2 in figure 4.12(a), the remapping can lead to a performance speedup. In R_2, 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 the smallest performance degradation. For the faulty n_1 scenario, all of the NMS-A/B/C heuristics yield the remapping R_2 which is the optimal decision. For the faulty n_2 scenario, it yields 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 figures 4.12(a) and 4.12(b), the analytical model estimates the degradation within

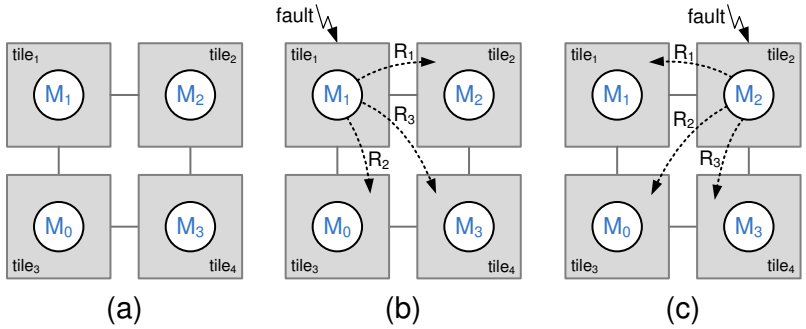


Figure 4.11. Initial mapping and the two single fault scenarios showing all possible remappings.

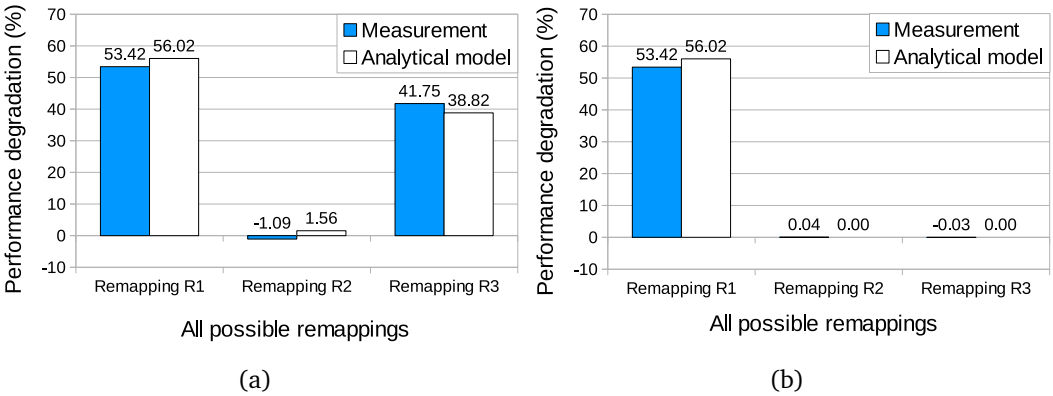


Figure 4.12. Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty (a) and when n_2 is faulty (b) as shown in figures 4.11(b) and 4.11(c), respectively.

3% of the measured values. The inaccuracy of the analytical model is due to the latency introduced by the communication API (see section 3.2.2) and the unaccounted context switching times when several tasks are running on a processor.

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 figure 4.13(a). 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 figures 4.13(b) and 4.13(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

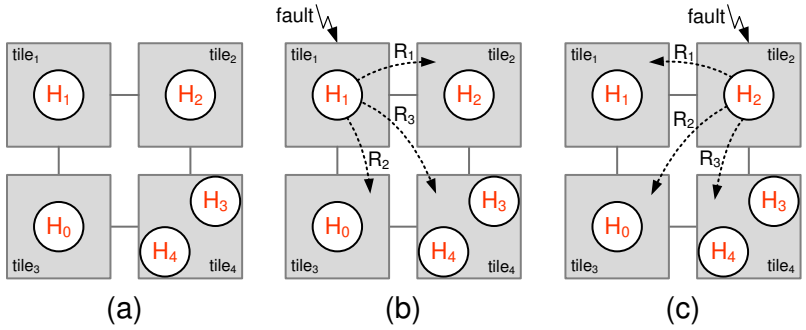


Figure 4.13. Initial mapping and the two single fault scenarios showing all possible remappings.

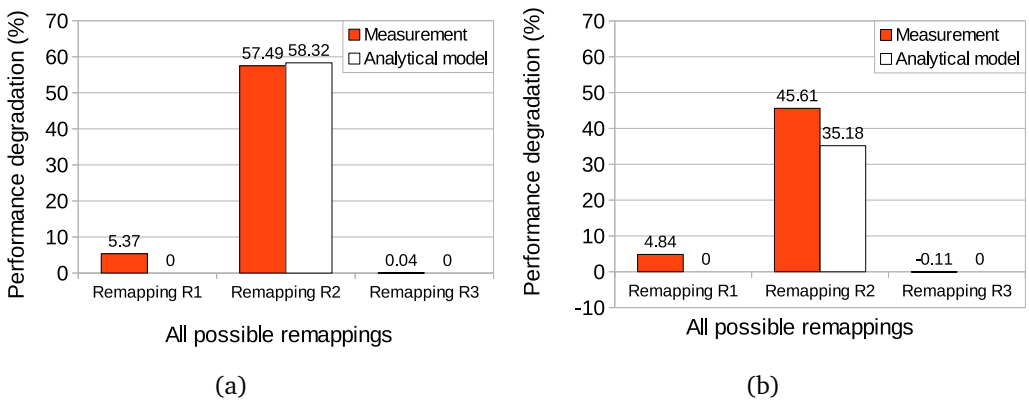


Figure 4.14. Comparison of measured and calculated performance degradation of all possible remappings when n_1 is faulty (a) and n_2 is faulty (b) as shown in figures 4.13(b) and 4.13(c).

in figure 4.14(a). 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 figure 4.14(b). 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.

To evaluate the computation time of the heuristics, the two remapping scenarios given in figures 4.11 and 4.13 are used for M-JPEG and H.264 applications. The NMS-A/B/C heuristics, which aim at minimizing the throughput degradation, are implemented on the platform with some optimizations such as static memory allocation (as opposed to the results reported in figure 4.8 which used dynamic memory allocation). Their computation times are displayed in table 4.8.

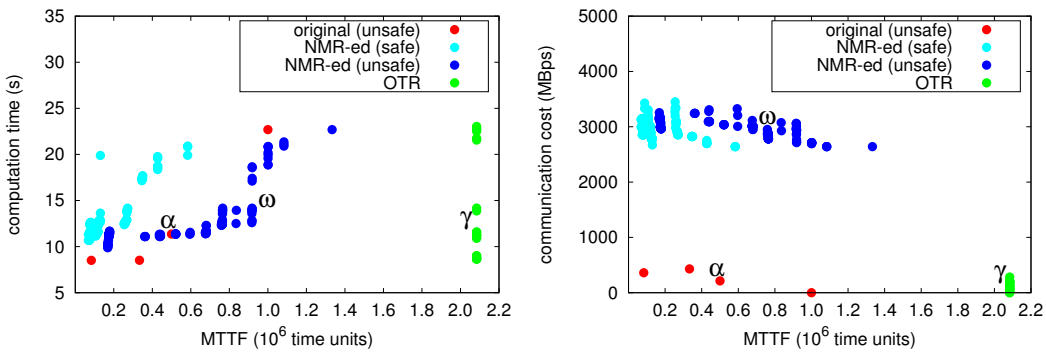
Table 4.8. Computation times of remapping heuristics

Method	**Avg. execution time (c.c.)**	
	M-JPEG	H.264
NMS-A	8198	8172
NMS-B	19608	19603
NMS-C	6403	6664

4.5.5 Reliability evaluation

In order to exemplify the use of our reliability estimation technique on a case study, we use once more the MPEG-2 decoder application described in section 4.5.1. We consider again an input video of 15 seconds long with a resolution of 704 × 576 pixels and the throughput goal is to achieve a frame rate of 25 fps. The XY-routing-based NoC architecture shown in figure 4.2(b) is considered. It includes RISC and DSP cores, both capable of executing all the tasks of the application. NoC links are assumed to provide a bandwidth of 1 GBps. T_{cap}^{CT} and d are given in tables 4.2 and 4.3 respectively. The NoC platform has a fork-voter non-programmable core at n_5. The fork and voter tasks are assumed to execute much faster than the tasks of the application, thus, fork-voter node is not the critical node in terms of throughput. However, since all fork and voter tasks are mapped on this single fork-voter core, the links connected to the fork-voter node are expected to be more overloaded than the rest of the links. Moreover, the fork-voter core is assumed to have a much smaller failure rate. Since it is a simple core, it can indeed be designed in a fault-tolerant manner and can occupy a small area. For the RISC and DSP processor cores, we assume constant failure rates equal to 10^{-5} and 10^{-6}, respectively. For simplification purposes, these rates are taken as constants although failure rates are known to increase with wear-out [Huang, Yuan and Xu, 2011]. Note that the approach would allow also incorporating a time-varying failure rate function $\lambda(t)$ into our model. Adopting such a failure rate would not be essential to the present work, so we chose not to experiment with it and leave the extension to a future work.

In the case study, we make use of the NMR-per-single-task strategy and apply the TMR pattern to all the tasks of the MPEG-2 decoder giving us a task graph with 64 tasks. Using our genetic algorithm based mapper tool, we computed the Pareto mappings of the described case study with respect to the throughput, MTTF and communication objectives as shown in figure 4.15 for the original task graph, NMR-ed task graph (in the case of safe and unsafe failure) and the online task remapping technique. As the original task graph is not redundant, its MTTF



(a) Computation time vs. MTTF (b) Communication cost vs. MTTF

Figure 4.15. Comparison of the Pareto points of original and NMR-ed task graphs as well as the OTR design points

values are obtained using the model for unsafe failure.

Considering the case of unsafe failure, it can be seen that the NMR pattern leads to a number of new Pareto mappings with better MTTF values for the same or better throughput values than the original task graph. In the case of the original task graph, the design point that satisfies the throughput constraint of 25 fps in 15 seconds and that has the maximum MTTF value is at $(0.5 \times 10^6, 11.36, 214)$ and marked as α. The Pareto point that would be chosen after the NMR transformation is at $(0.92 \times 10^6, 14.09, 2947.4)$ and marked as ω. Both of these points satisfy the throughput constraint but ω is 84% more reliable than α.

Figure 4.15(b) shows the same design points in figure 4.15(a) plotted with respect to the communication cost and MTTF. It can be seen that the communication costs for NMR-ed task graphs are significantly higher than the original. In the case of ω and α, the difference is by an order of magnitude.

If online task remapping is used, MTTF is calculated as 2083707 $time\ units$. The performance cost of supporting the online task remapping technique is twofold. On the one hand, the total execution time of each core will be increased by the number of invocations of the self-testing routine times the execution time of a single invocation of the self-testing routine. A second contribution to the overhead is due to the additional code executed in the task bodies even in absence of faults in order to support fault recovery. However, our experiments on the real platform show that the performance overhead can be as small as 1.5% (see section 5.5). Therefore, for the same mapping represented by α, the design point given by OTR, which is shown in figures 4.15(a) and 4.15(b) by γ, is at $(2.08 \times 10^6, 11.53, 214)$. The OTR design points plotted in figure 4.15 correspond to

different initial mappings. With optimal remapping decisions, the performance is expected to degrade along those points during the lifetime of the application as the nodes become faulty.

In the case of safe failure, NMR serves to detect and possibly mask transient and permanent faults. Instead OTR, which uses self-testing as the fault detection techique, tolerates only permanent faults and requires the application to allow limited error propagation until the recovery is completed. If tolerance to permanent faults is aimed at and benchmark applications allow limited error propagation, as it is the case in this thesis, OTR outperforms NMR by leading to better design points for all three metrics as shown in figure 4.15 (NMR-ed (safe) vs OTR). OTR yields an MTTF value that is at least two times more than that of TMR (safe). Even if NMR is considered with unsafe failure, which does not guarantee correct results to be produced by the application, OTR keeps outperforming NMR (NMR-ed (unsafe) vs OTR). The MMTF of OTR is almost 50% more than the maximum MTTF that can be achieved with TMR (unsafe).

4.6 Summary

We formulated the optimal task mapping problem for NoC-based multiprocessors with generic topologies and deterministic routing as an integer linear programming (ILP) problem with the objective of minimizing the communication traffic in the system and the total execution time of the application. We used it to obtain optimal task remappings in presence of faults in processing cores. Several heuristics have been proposed and their results have been compared with respect to the optimal remappings. Our results showed that LNMS heuristics provide a parameterized solution that can accommodate different trade-offs. Comparing the results of heuristics with the optimal solutions, we have found that they are in average within 7% proximity in terms of degradation distance.

We also showed with two real-life case studies that the NMS heuristics are able to find near-optimal remappings. Moreover, the experimental results show that the overhead in terms of execution time due to the execution of the remapping heuristics is almost negligible compared to the execution time of the whole application. This means that the proposed heuristics can be used in a fault recovery mechanism without a substantial impact on the user experience.

Our solution has been demonstrated with use-case applications for several fault scenarios on a mesh-based platform adopting an XY-routing strategy. However the proposed solution can be applied to platforms with any topology and any deterministic routing strategy.

In dynamic environments where the applications that will execute on the platform are not a-priori known, overwhelmingly the case for mobile platforms, the approach where remapping decisions are made offline and for which exact degradation profiles are calculated a priori, cannot be applied anymore. Therefore we have to resort to online remapping decisions. Unfortunately the degradation bounds in such a case cannot be guaranteed and can be beyond the tolerances of the application. This calls for self-adaptive systems with adaptation capabilities that go beyond run-time task mapping, thus allowing the compensation of the degradation through other mechanisms such as the self-adaptation of application level parameters as described in chapter 6.

NMR is a concurrent self-checking technique for fault tolerance that can be applied at the KPN level. Our results show that NMR can turn the original application into a more reliable one and achieve the same throughput levels. However, it comes at the expense of a huge overhead in terms of the communication taking place in the network. Although we have not evaluated different number and placement of fork-voter cores, it can be argued that increasing the number of fork-voter cores can help to reduce the communication overhead of NMR-ed applications.

On the other hand, online task remapping is a much cheaper technique capable of yielding to a two-fold reliability increase compared to NMR, at the expense of a small overhead due to the fault detection technique (such as the execution of an online self-testing routine) and the fault recovery support via task remapping.

Chapter 5

Recovery Support in the Fault-aware Run-time Environment

Fault recovery is the stage starting from the detection of the fault till the point in which the application continues operating on fault-free nodes. In this chapter we focus on the description of the mechanisms that are defined for achieving fault recovery via run-time migration of processes among tiles. With the implemented fault recovery support, we aim to show the feasibility of the fault-aware run-time environment approach proposed in this dissertation. The results presented in this chapter have been published partially in [Derin, Cannella, Tuveri, Meloni, Stefanov, Fiorin, Raffo and Sami, 2013; Cannella et al., 2012].

The remainder of this chapter is organized as follows. Section 5.1 discusses the contributions of this dissertation with respect to the state-of-the-art. Sections 5.2 and 5.3 present the two techniques proposed for fault recovery based on fine-grained checkpointing-and-rollback (CRR) and roll-forward (RFR), respectively. Each section details the required changes at the application, RTE and hardware layers. Section 5.4 presents results obtained experimentally on the platform and compares the two techniques in terms of their overhead in time and area.

5.1 Contributions with respect to the state of the art

The previous studies on task migration have some shortcomings and are not applicable in our fault recovery scenarios. First of all, the approaches proposed in [Bertozzi et al., 2006] and [Acquaviva et al., 2008] are for shared memory systems which makes them inapplicable for purely distributed memory platforms such as ours. Secondly, even when a task migration approach is proposed for

purely distributed memory multiprocessors such as the one proposed in [Almeida et al., 2009], it cannot be sufficient by itself to recover from faults. Because additional mechanisms should be put in place to manage fault recovery, for example, by coordinating a global checkpoint and rolling back to it. What to checkpoint in the case of fault recovery and of non-fault related task migration differs significantly. In the task migration case, checkpointing the state of the task is sufficient. Instead, in the fault recovery case, a snapshot of all the tasks should be checkpointed.

Thirdly and most importantly, task migration techniques mentioned above are all purely software-based. Execution of such techniques involves operations performed by the processor that hosts the tasks to be migrated. Obviously, in the fault recovery case, that processor is faulty and cannot be trusted with the remapping of the tasks. Instead, in our approach, we introduce a task migration hardware module that carries out the basic funtionality required by the faulty processor. Proactive fault management techniques [Salfner et al., 2010; Lan and Li, 2008] would not require such a task migration hardware module because tasks, which are running on a core that is likely to fail during the next time window, would be migrated before the fault occurs. However, to this date, there is no perfect failure predictor that achieves correct prediction of all failures (i.e., a recall rate of 1) with reasonable precision. Furthermore, most proactive fault management techniques [Salfner et al., 2010] are for large-scale processing systems, and they cannot be adopted in a straight-forward manner for SoCs without statistical data and models.

If migration is to be supported in a programming model agnostic manner, the state to be migrated consists of the processor state with all of its internal registers' values and the application state at the migration point stored in the memory. This state should be fully transfered to the new processor. When migration is done between processors of different types, the state should also be adapted for the new processor. The size of the transferred state as well as the need to adapt it brings an overhead to the migration procedure. By adopting PPN as the model of computation in our approach, this overhead is partly avoided because the state of a PPN process is known independently from the application at the beginning of the iterated process body. This is, in fact, exploited by the proposed techniques described in this chapter. The approach presented in [Stralen and Pimentel, 2012] is similar in that sense. However, the memory requirement of the global checkpoint, which involves storing all of the tokens processed by a task in between two checkpoints, makes this approach too prohibitive to be adopted in our platform. Therefore, in one of our recovery techniques, we resort to fine-grained checkpointing and rollback which can also be referred to as re-execution

[Kang et al., 2014a; Izosimov et al., 2012].

As mentioned in the previous chapter, there has been some work done on system-level fault tolerance in NoC-based platforms based on spare cores or task remapping. Some of those techniques propose the allocation of spare cores [Chou and Marculescu, 2011; Ababei and Katti, 2009; Khalili and Zarandi, 2012], while others investigate fault-aware remapping on fault-free cores of tasks running on faulty processing elements [Lee et al., 2010]. However, none of these techniques deal with the recovery problem and they only address the remapping problem (i.e., the selection of the new cores that execute the tasks of the faulty core) without an actual implementation on a real platform.

Our work in this chapter is fundamentally novel in the aspect that it realizes fault-aware run-time management on an NoC-based platform and demonstrates two fault recovery techniques by reporting their results.

Our contributions in this field are

- a fault recovery technique that allows limited error propagation by checkpointing at the beginning of each iteration and rolling back to the beginning of the iteration in case of fault detection.
- a fault recovery technique that allows limited error propagation by rolling forward to a future point in the processed stream.
- implementation of the techniques on the platform by incorporating the required changes in the application, RTE and the tile architecture.
- evaluation and comparison of the techniques in terms of their steady-state performance overhead, recovery time and area overhead.

5.2 CRR: Fine-grained checkpointing and rollback based fault recovery

The proposed solution encompasses support for 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 fine-grained checkpointing and rollback based fault recovery technique (CRR) 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 the task migration hardware (TMH), which is responsible for notifying the run-time manager (RM) 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). The way that we have implemented fine-grained checkpointing guarantees recovery with limited error propagation for the majority of faults described in section 3.3.1 excluding those that lead to a wrong number of tokens in input or output FIFOs (E1.2). Alternatively, a complete hardware-level checkpointing of the NI buffer, input/output FIFOs and the iterators as well as a write operation implemented in hardware which is called directly by the STM upon a successful self-test could cover all possible corruptions. However, the additional overhead of such a solution would be an overkill for the small extra coverage that would be gained. It is a trade-off we have made between performance and fault coverage. In the remainder of this section, the fault recovery mechanism is explained in detail by describing the changes done at the application, run-time and hardware levels.

5.2.1 Modifications to the PPN processes

A part of the fault tolerance support involves the modification of process bodies. Algorithm 8 shows how the basic process body shown in algorithm 7 is modified to support the fault recovery mechanism.

Algorithm 7 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 7). 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 8 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 8, 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 7, i.e., a function that does not possess any hidden variable that depends 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 8, 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 7 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 processor hangs (e.g., due to infinite loops or stuck program counter), 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 crash error to the TMH module by raising the fault_detected signal.

The faults can be more subtle and may result in computational errors, which are known as silent data corruptions. 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 expected, it is detected by the self-testing module, which in turn signals the fault detection to the TMH.

If a crash occurs between the end of the 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, it can be argued 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.

5.2.2 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 in 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 RM calculates the new mapping of the tasks using the remapping heuristics (see chapter 4). 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 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.

Decentralization of the Run-time Manager

Centralized techniques represent a single point of failure and thus they should be avoided in fault tolerance mechanisms. The 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 act as the RM when it receives an interrupting fault detection message. The closest fault-free tile is chosen as the RM of a tile. The TMH

of a faulty tile sends the fault detection message to the RM instance assigned to its tile. Given the single fault assumption (see section 3.3.1), when a fault occurs, every fault-free tile is informed about the faulty tile and updates its 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-select their RM.

5.2.3 Task migration hardware module

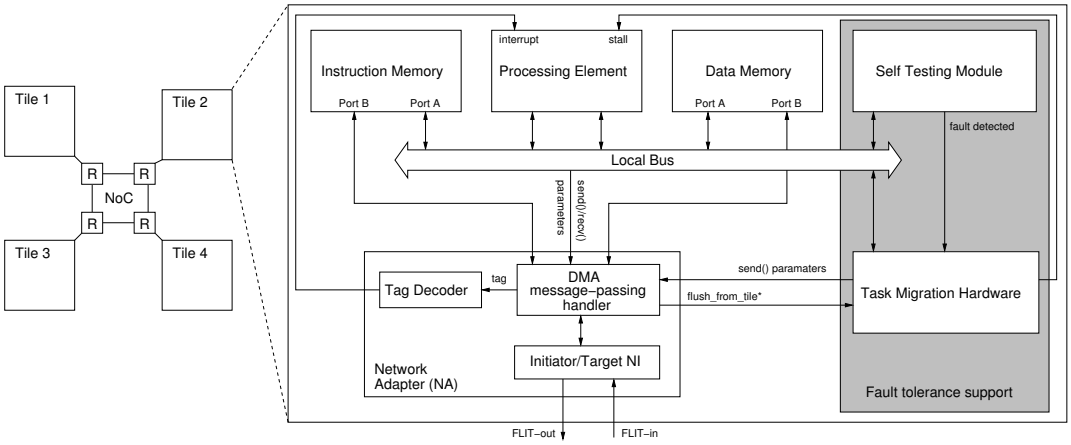
As shown in figure 5.1(a), the base NoC tile architecture, composed of the processing element (PE), local memories, and Network Adapter (NA), is extended with two hardware modules supporting the implemented fault tolerance technique: the *Self-Testing Module* (STM) and the *Task Migration Hardware* module (TMH).

As explained in section 3.3.2, the STM (shown in grey in figure 5.1(a)) supports the execution of testing routines in the processing element of each NoC tile. At the end of the execution of the software routine, the STM checks whether the signature of the outputs of the routine (calculated by using a CRC algorithm) matches the one previously stored in one of its registers, activating the *fault_detected* signal connected to the TMH in the case of a negative answer.

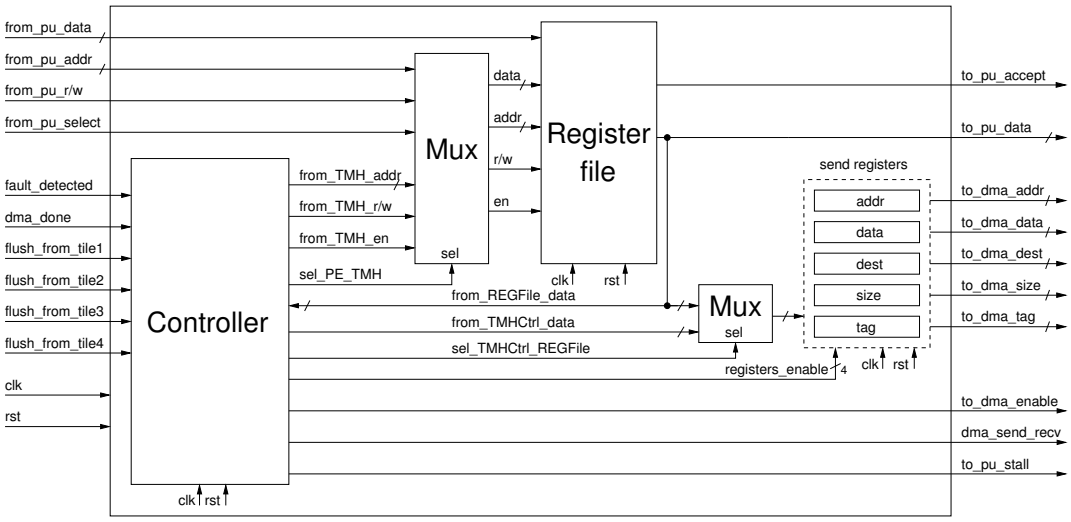
The task migration hardware (TMH) module is mainly responsible for extracting the critical data from the faulty tile. As shown in figure 5.1(a), the TMH resides alongside the Network Adapter of each tile. It receives as input a fault detection signal from the STM. Upon the detection of the fault, the 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, thus guaranteeing 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.



(a) Fault tolerant tile



(b) Task migration hardware module

Figure 5.1. Interfaces and internal block diagrams

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 that of the processing core.

The interface and the internal block diagram of the TMH are shown in figure 5.1(b). 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 *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 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 each take only a clock cycle in order to reduce the overhead of the update operation. After fault detection, TMH carries out a number of *send* operations by using the programmable DMA to notify the RM, and sending states of mapped tasks and data of mapped channels.

5.3 RFR: Roll-forward fault recovery

In streaming applications that conform to the application model described in section 3.2.1, the stream is composed of a sequence of stream units with increasing indices. The main idea behind the technique proposed in this section is to perform a roll-forward to the next stream unit upon the detection of the fault in the processor. At the time of fault detection, the stream unit will be processed partially and an incomplete output will be produced. When, eventually, the recovery is completed and the following stream unit is processed, the output will start being error-free as expected. Therefore, the proposed mechanism is applicable only if, for a limited time, such incomplete or incorrect output is allowed. However, this is typically the case of multimedia streaming applications, in which the loss of few stream units does not significantly influence user experience, or of non-critical sensing applications, in which processing of a sample can be omitted. The RFR technique guarantees recovery with limited error propagation against all kinds of corruptions described in section 3.3.1. Fault tolerance consists of fault detection and recovery phases. The fault-tolerance mechanism

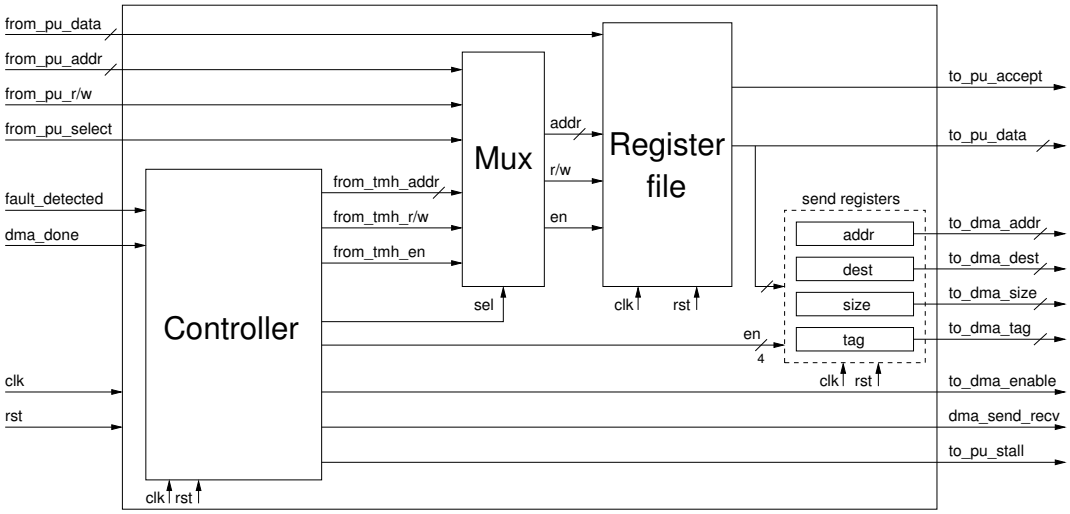


Figure 5.2. Block diagram of the task migration hardware

relies on online periodical software-based self-testing and a self-testing module for detecting faults [Gizopoulos, April-June 2009]: incorrect results would propagate to the user until the next invocation of the self-testing routine that detects the fault and initiates the recovery. The proposed recovery technique, which is the main focus of this section, involves the implementation of a fault-aware distributed run-time environment, some modifications to the PPN process template and addition of a task migration hardware module into the tile architecture of the NoC.

5.3.1 Task migration hardware module

The fault tolerant tile architecture for the RFR scheme is similar to the one of CRR shown in figure 5.1(a) with slight differences in the STM and major differences in the TMH modules.

Differently from the STM described in section 3.3.2, in the RFR case, the timer works in a periodical fashion. The period, which is equal to the self-testing period, is written once at boot time into the *slv_timer_limit* register. It is started only once via the *slv_timer_start* register. Then the timer counts down. At the time it reaches zero, a check is done by monitoring the *start_stop* input of the timer whether there has been a self-testing routine execution during the count down. If so, the timer is reset again to the *slv_timer_limit* value and starts counting down again. Otherwise, a crash error is assumed and *fault_detected* signal is

raised.

The TMH is responsible for isolating the processor (in order to avoid further error propagation) and for notifying, as explained in detail in section 5.3.2, the *run-time manager* (RM), in charge of managing the task remapping procedure. The structure and the function of TMH in the case of RFR are much simpler than the case of CRR because it is only responsible for isolating the processor when the fault is detected, then reporting the fault to the run-time manager through an interrupting message. Figure 5.2 shows the internal structure of TMH which is similar to that of CRR but the size of the register file is smaller. The TMH uses four internal registers (address, destination, size and tag) which corresponds to the parameters of the *send()* message passing primitive of the platform . There are no *flush_from_tile** signals since there are no flush messages to receive. In addition, there is a *send* data register, which stores the memory address of the fault detection message to be sent, as the fault detection message is not required to be stored inside the TMH.

Smaller area of the STM and TMH modules as well as their less frequent use with respect to the processor decreases the likelihood of their failure before the processor. Nevertheless, the STM and TMH modules should be hardened against faults in order to avoid single points of failure.

5.3.2 Fault-aware remapping support

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

When the RM receives the fault detection message from the TMH, the tasks on the fault-free PEs would be either executing their processing functions or being blocked on a read or a write. Some tokens of the tasks could be waiting to be sent on the software FIFOs or to be received in the NI buffer, or even travelling in the NoC. Similarly, pending requests, which are sent from consuming tasks to the producer tasks, might have been already received or might be still travelling

along the NoC. Given such possibilities, at fault detection the RM takes a number of steps to flush the current state from all the tiles and makes the tasks ready to continue executing the next stream unit.

Firstly, it sends a *FLUSH_TASK* interrupting message to the tile hosting the source task. Upon receiving this interrupt, the interrupt handler of the source tile sets a flag (*isTaskToBeFlushed*) requesting the flushing of the source task. The process bodies and PPN communication primitives are modified as explained in section 5.3.3 to respond to such requests. In the special case of the source task, the source seeks the stream forward to the next stream unit and responds back to the RM with the index of the next stream unit (*next_stream_unit_index*). The RM continues sending the *FLUSH_TASK(next_stream_unit_index)* interrupting messages to all other tiles that run at least one task (except the faulty tile). This interrupt message sets the *isTaskToBeFlushed* flag for all the tasks on the tile. It also marks the iterator values to be updated (*isStreamUnitIndexToBeUpdated*) when the tasks are resumed after the recovery is finished. The purpose of flushing all tasks is to delete the state of the current stream unit by removing the tokens in all input and output FIFO channels and by resetting pending requests. However, serving the flush request is not straightforward due to tokens or requests still travelling on the NoC. Therefore, a *channel flush mechanism* is employed in order to guarantee that all the state (tokens and requests) are received by the tiles and the NoC buffers are emptied of such information.

The channel flush mechanism is based on the idea that the NoC transmits the packets in order. Therefore, if a special *CHANNEL_FLUSH* token is sent by a source tile and is received by a destination tile, it is guaranteed that any data that have been sent by the source tile before the special token would have been already received in the NI buffer of the destination tile. Given a PPN task graph, the channel flush mechanism should guarantee that at least one *CHANNEL_FLUSH* token is sent from the source tiles to the destination tiles for every channel of the PPN task graph (except those channels in/from/to the faulty PE and those that are inside the same tile, i.e., source and destinations tile are the same).

Algorithm 9 shows the channel flush mechanism carried out by each PPN process as a part of serving the flush request for the task (*serveFlushRequest()*). In lines 3–5, firstly, each task sends a *CHANNEL_FLUSH* token to each destination tile of its output channels, given that the destination tile is not the faulty tile and that the destination tile is not the same tile that runs the task. In lines 6–13, each task receives a *CHANNEL_FLUSH* token from each source tile of its input channels, given that the source tile is not the faulty tile and that the source tile is different than the tile of the task. In order to let other tasks on the same tile execute their functions during the channel flush process, the task relinquishes

Algorithm 9 The channel flush mechanism inside *serveFlushRequest()* of task t

```
1: CH_I(t) : input channels of task t
2: CH_O(t) : output channels of task t
3: for all ch ∈ CH_O(t) and cons(ch) is not faulty and prod(ch) ≠ cons(ch) do
4:     send(token, 1, cons(ch), CHANNEL_FLUSH_TAG);
5: end for
6: while all CHANNEL_FLUSH messages are not received do
7:     for all ch ∈ CH_I(t) and prod(ch) is not faulty and prod(ch) ≠ cons(ch) and !isFlushed(ch)
       do
8:         if nonblocking_recv(token, 1, prod(ch), CHANNEL_FLUSH_TAG) then
9:             isFlushed(ch) = true;
10:        end if
11:    end for
12:    yield();
13: end while
14: return
```

the processor by calling *yield()* while polling on the *CHANNEL_FLUSH* tokens.

In the case of requests, the flushing procedure is a bit different due to the fact that the requests are sent in the reverse direction of a channel (i.e., from the consuming task to the producing task). The channel flush mechanism should also make sure that a *CHANNEL_FLUSH* token is sent in the reverse direction of every channel. This can be achieved by slightly modifying the algorithm in algorithm 9 by replacing *prod* with *cons* and vice versa.

After the completion of the channel flush mechanism, the task flushing operation for each task continues by transferring the tokens in the NI buffer to the FIFO channels and by clearing all the tokens in the FIFO channels as well as the requests in the NI buffer. As a result, the resources used by the current stream unit are recovered and no residual state remains which would likely cause negative impacts in the future operations. Finally, each task communicates to the RM that its task flushing operation has been completed and waits for a *resume* message from the RM before continuing execution.

Then, the RM calculates the new mapping of the tasks on the faulty PE using a remapping heuristic presented in section 4.3 Each tile has to store an up-to-date copy of the middleware table which is used to look up the tile of a task. Therefore, the RM updates the middleware table of all tiles except the faulty tiles. Then, it activates the tasks on their new processors. Finally, it sends all other tasks the *resume* messages to let them continue their execution.

The decentralization of the RM as explained in 5.2.2 is also valid for RFR.

Figure 5.3. The modified read(token, channelID) primitive

```
 1: if fifo[channelID] is empty then
 2:     sendRequest(channelID);
 3: end if
 4: while fifo[channelID] is empty do
 5:     if serveFlushRequest() then
 6:         return
 7:     end if
 8:     process_NI_msgs();
 9: end while
10: fifoGet(token, channelID);
11: return
```

5.3.3 Modifications to the PPN processes

As a part of the fault recovery support, the process bodies and the PPN communication primitives have to be modified. Algorithm 10 shows how the basic process body template shown in algorithm 7 is modified.

In the case that a process is activated on the new tile, the checking of migration flag (line 1) will allow the process to start execution from the correct stream unit index.

In the case that a process is not on the faulty tile, it receives a *FLUSH_TASK* request. This request is eventually checked in the modified *read()* primitive (in figure 5.3 line 5) or in the modified *write()* primitive (in figure 5.4 line 2). The *serveFlushRequest()* function carries out the channel flush mechanism by clearing the FIFO channels and requests as explained in section 5.3.2. It also sets *isStreamUnitIndexToBeUpdated* and yields in a loop until the *resume* message is received. If a flush request is served inside *serveFlushRequest()*, it returns true. Therefore, the blocking read (in algorithm 10 line 10) or blocking write (in algorithm 10 line 15) calls return immediately after the flushing of the task, thanks to the modifications in figure 5.3 line 6 and figure 5.4 line 3. The check on *isStreamUnitIndexToBeUpdated* in algorithm 10 line 11 and 16 allows the process to break execution of the current stream unit and reach line 5 where it starts the execution of a new stream unit with the updated stream unit index.

5.4 Experimental results for CRR

In this section, we describe a set of experiments that we performed in order to evaluate the implemented CRR scheme. The application case studies are de-

Algorithm 10 The modified PPN process template

```
1: if migration then
2:     i0 = newStreamUnitIndex;
3: end if
4: for (i=i0 ; i <M; i++) do
5:     if isStreamUnitIndexToBeUpdated then
6:         i0 = newStreamUnitIndex;
7:         isStreamUnitIndexToBeUpdated = false;
8:     end if
9:     for (j=0 ; j <N; j++) do
10:        read(in, CH1);
11:        if isStreamUnitIndexToBeUpdated then
12:            break;
13:        end if
14:        out = f(in);
15:        write(out, CH2);
16:        if isStreamUnitIndexToBeUpdated then
17:            break;
18:        end if
19:    end for
20: end for
21: return
```

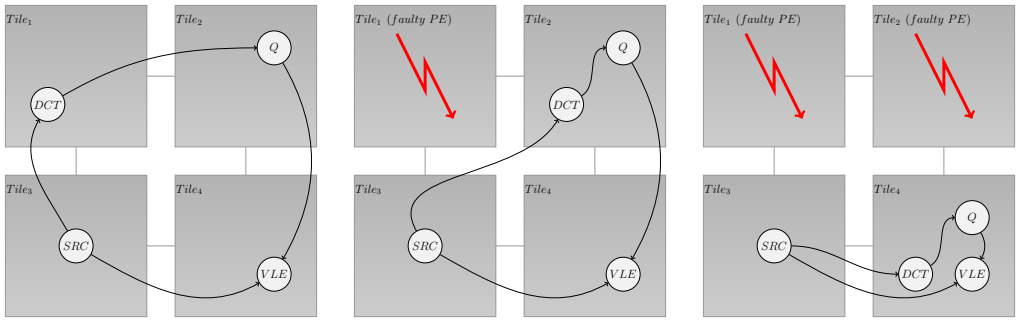
Figure 5.4. The modified write(token, channelID) primitive

```
1: while fifo[channelID] is full do
2:     if serveFlushRequest() then
3:         return
4:     end if
5:     process_NI_msgs();
6: end while
7: fifoPut(token, fifo[channelID] );
8: process_NI_msgs();
9: return
```

scribed in section 4.5.3. We map these applications onto a 2×2 mesh of general-purpose processors, as detailed in section 3.1, implemented on a Virtex-6 FPGA board. We present a remapping process, exploiting the fault recovery mechanism described in section 5.2.

5.4.1 Fault recovery time overhead

To assess the performance of the fault recovery mechanism, we evaluated it with a single fault scenario (initial mapping shown in figure 5.5(a) and the fault on



(a) Initial mapping (b) First fault (c) Second fault

Figure 5.5. A fault scenario example

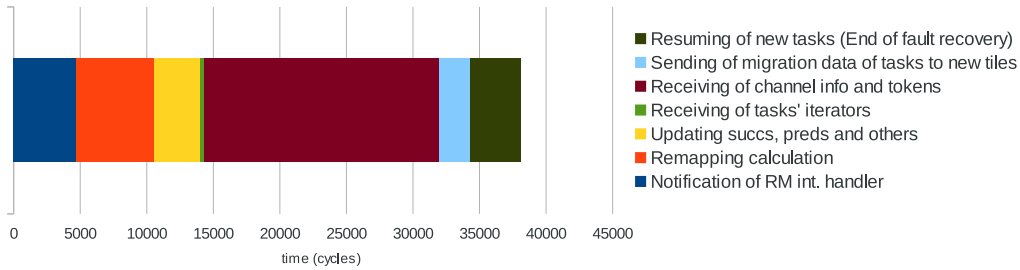


Figure 5.6. Execution times of fault recovery actions

$tile_1$ as shown in figure 5.5(b).

Figure 5.6 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 transfering the state to the new tile does not take as much time. It is worth noting 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

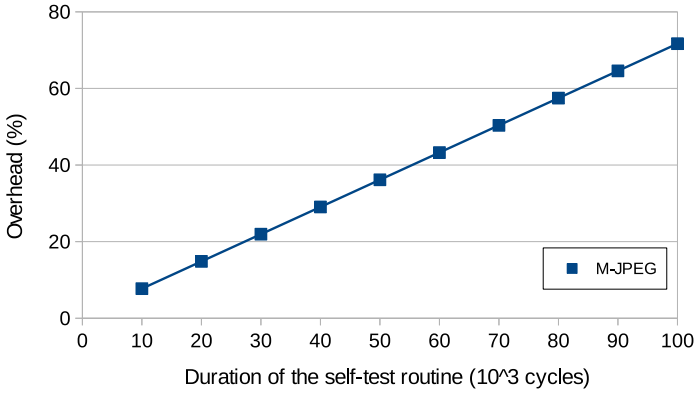


Figure 5.7. Performance overhead with respect to the duration of the self-testing routine

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. The results also reveal that the execution time of the remapping heuristic constitutes a relatively small portion of the fault recovery time.

5.4.2 Steady-state performance overhead

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 10k to 100k cycles. The mapping used in the calculations is the one of figure 4.11(a). As shown in figure 5.7, 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 40k cycles as a typical duration for the self-test (taken from [Gizopoulos, April-June 2009] 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.

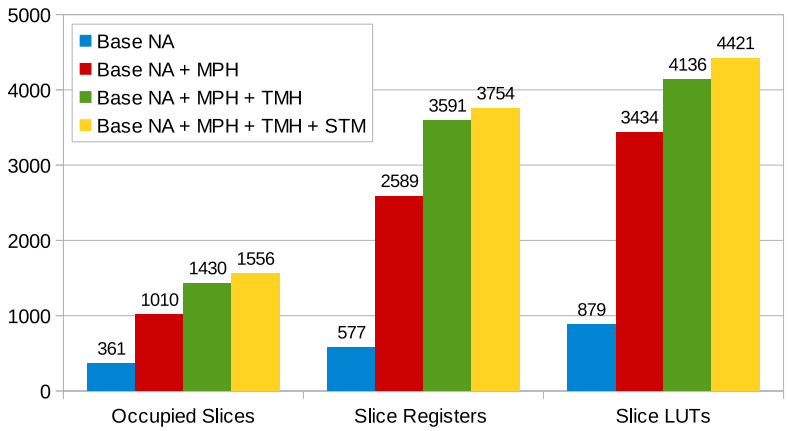


Figure 5.8. Area occupation overhead in comparison to the baseline network adapter due to the support for system adaptivity and fault-tolerance

5.4.3 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, 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 figures 5.8 and 5.9, 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 prototyping 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 the MPH. During

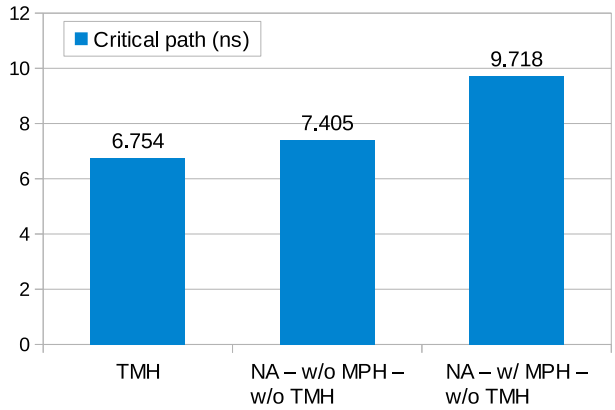


Figure 5.9. Critical path length overhead related with support for system adaptivity and fault-tolerance

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 figure 5.10. 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 include also the area of the memory modules in the calculation of the baseline tile area, not accounted for in the presented plot.

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 M-JPEG 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. In order to see the change in the area, we have synthesized the TMH using Xilinx ISE for different number of channels (i.e., 4, 8, 16, 32). Figure 5.11 reports occupied slices (representative of total area), slice registers (representative of memory) and slice LUTs (representative of combinational logic). The area of TMH increases almost linearly with increasing number of channels mainly due

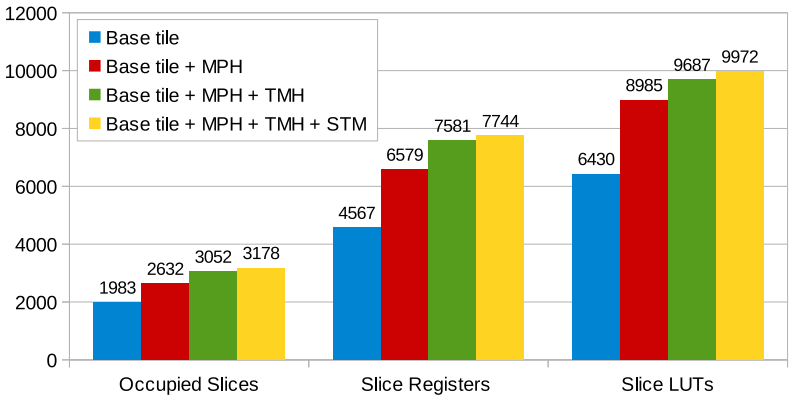


Figure 5.10. Area occupation overhead in comparison to the baseline tile architecture due to the support for system adaptivity and fault-tolerance

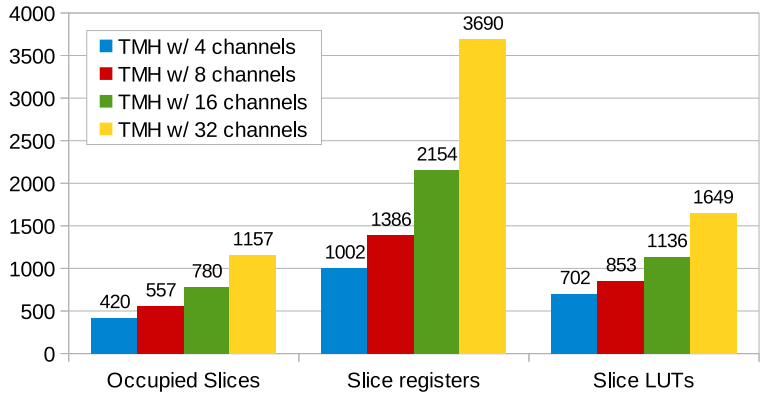


Figure 5.11. TMH area for varying number of supported channels

to the increasing size of the register file. This could hinder the adoption of the rollback based recovery scheme for applications with high number of channels.

5.5 Experimental results for RFR

We tested the proposed RFR technique in the case of an M-JPEG encoder and an H.264 decoder application mapped on the 2×2 NoC platform with Microblaze as processing elements. The roll-forward position for M-JPEG is the beginning of the next frame, whereas, in H.264, it is the beginning of the next I-frame.

Table 5.1. M-JPEG fault scenarios

scenario	initial mapping (DCT, Q)	1st fault	2nd fault
1	(tile1, tile2)	tile1	tile2
2	(tile2, tile1)	tile1	tile2
3	(tile1, tile2)	tile2	tile1
4	(tile2, tile1)	tile2	tile1
5	(tile2, tile2)	tile2	tile1
6	(tile1, tile1)	tile1	tile2

Table 5.2. H-264 fault scenarios

scenario	initial mapping (IDCT, IntraPred, Deblock)	1st fault
1	(tile2, tile4, tile4)	tile2
2	(tile2, tile4, tile4)	tile4

5.5.1 Fault recovery time overhead

We report the results of different fault scenarios. A fault scenario is identified by the initial mapping and the order of fault injections on the different tiles. Our implementation is general in the sense that it does not restrict the initial mapping and fault scenarios. However, since the source and sink tasks are generally mapped to tiles that are connected to special I/O interfaces, their remapping is not feasible. Therefore, such tiles are excluded from fault scenarios. They should be hardened with lower level techniques. M-JPEG has four tasks. We consider the scenario in which the source task (*SRC*) is mapped to $tile_3$ and the sink task (*VLE*) to $tile_4$. Fault scenarios are obtained by considering all possible mappings of the *DCT* and *Q* tasks as well as all possible fault sequences on $tile_1$ and $tile_2$ as shown in table 5.1. The first fault scenario for M-JPEG is depicted in figure 5.5. Faults are injected under the assumption that the second fault is injected after recovery from the previous fault is completed. H.264 has five tasks. The source task (*GetData+Parser+Cavlc*) is mapped to $tile_1$. The sink task (*PrintMB*) is mapped to $tile_3$. Due to the memory limitations of the platform, only single fault scenarios on $tile_2$ and $tile_4$ are evaluated with the initial mapping of *IDCT* to $tile_2$; *IntraPred* and *Deblock* to $tile_4$ as shown in table 5.2.

Fault injections are simulated by activating the fault detection signal connected to the TMH directly with the PE. Figures 5.12 and 5.13 show the finishing times of each recovery action averaged over 10 different fault injection times for the same fault scenario for M-JPEG and H.264, respectively. Time 0 represents

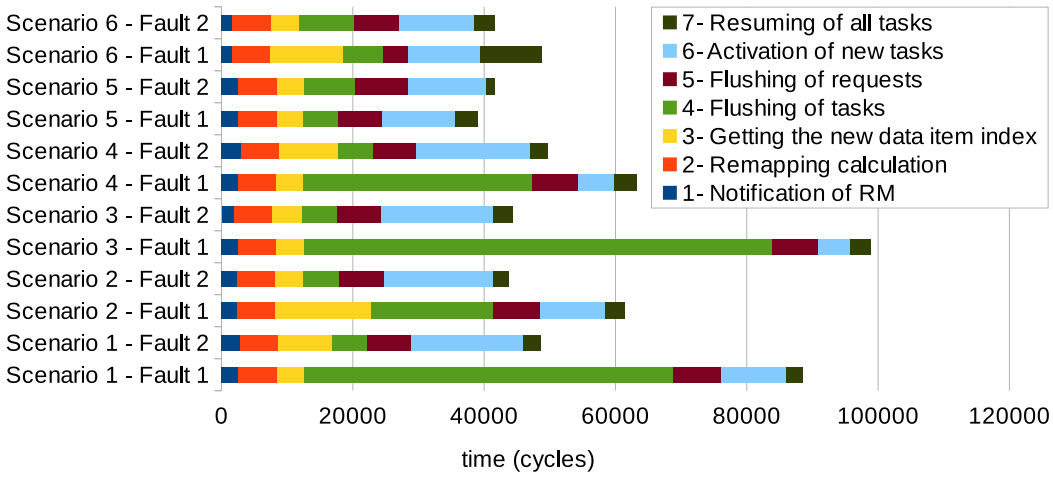


Figure 5.12. The time of fault recovery actions for M-JPEG

the instant in which the TMH is activated (i.e., fault detection instant). It can be seen that the flushing of tasks takes the majority of the time in the case of recovery from the first fault and its duration may vary greatly for each experiment with different fault injection times or different fault scenarios. This is mainly due to the fact that the task may be executing its processing function before it gets to serve the flush request. In the worst case, the flush request may arrive just after a *read()* call, leading to a delay as long as the duration of the processing function until the flush request is served in the next *write()* call. In the M-JPEG case, one iteration of the heaviest task, i.e. DCT, takes around 132*k* cycles.

If the fault injection is done at the hardware level, the time between the occurrence of the fault and its detection would have to be added to the fault recovery time. Since faults are detected via self-testing, the worst case fault detection time would be equal to the period of the self-testing (in case the fault occurs immediately after a successful self-testing). The shortest self-testing period for the M-JPEG case would be equal to the processing time of one frame. In the case that M-JPEG tasks are mapped one-to-one on the 2×2 platform, the execution time of one frame is around 10×10^6 clock cycles. The observed fault recovery times in figure 5.12 would constitute only a small fraction of the total recovery time. That is to say, the fault recovery time is fundamentally determined by the self-testing period. Same observation applies to H.264 as well.

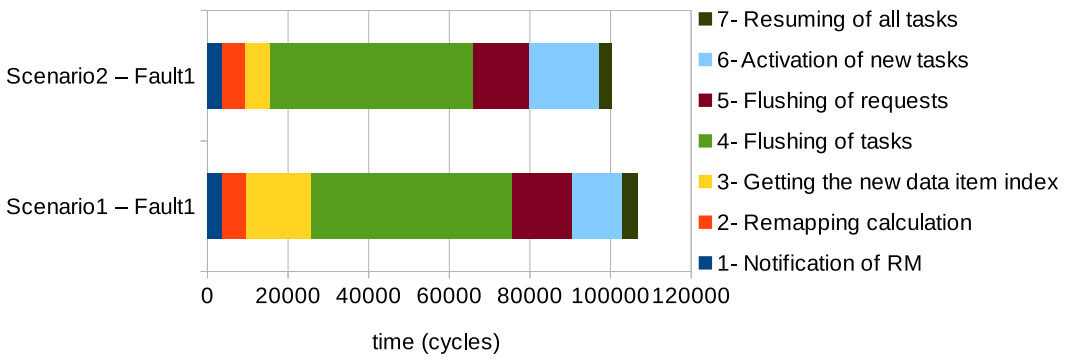
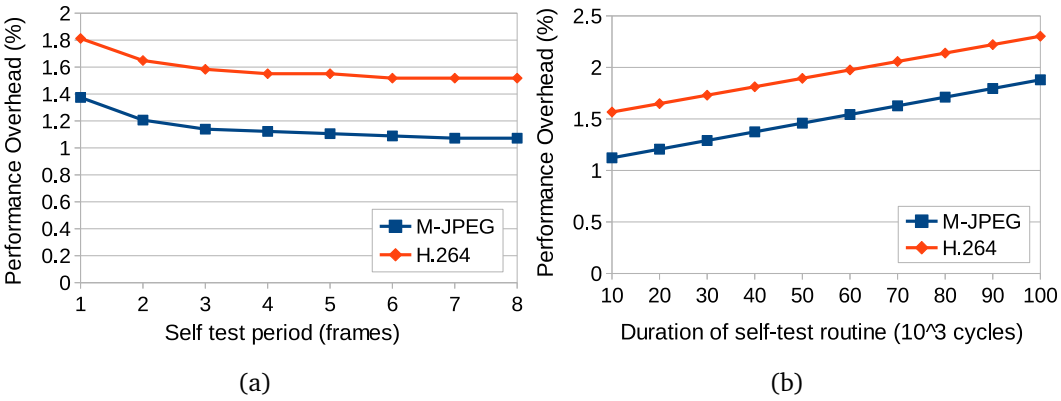


Figure 5.13. The time of fault recovery actions for H.264

5.5.2 Steady-state performance overhead

The self-testing routine can be executed at any desired frequency. Obviously, this frequency would affect the overhead of the technique during normal operation. Assuming a self-testing execution time of 40k cycles as a typical duration for the self-testing routine (taken from [Gizopoulos, April-June 2009] for a processor of supposedly similar complexity to Microblaze), the overhead with respect to varying self-testing period (quantified by the number of frames processed within the period) is shown in figure 5.14(a). It can be seen that the overhead due to the self-testing routine diminishes completely when the period is greater than 7 frames. The converged overhead value of 1.07% for M-JPEG and 1.51% for H.264 is due to the modifications done in the PPN processes to enable fault recovery.

Similarly, one can assess the overhead with respect to the duration of the self-testing routine. Given that a self-testing action is executed for every frame, the overhead by varying the self-testing duration is shown in figure 5.14(b). Increasing the execution time of the self-testing routine by an order of magnitude (from 10k to 100k cycles) increases the overhead only slightly, from 1.1% to 1.9%. This is a much lower overhead compared to the CRR-based technique presented in section 5.2 for which it changes from 7.7% to 71% for the same range of self-testing durations due to the fact that they do the self-testing for every iteration of a task body. Considering the self-testing duration as 40k cycles, we see that the performance overhead would be 1.37% for M-JPEG and 1.81% for H.264.



(a) (b)

Figure 5.14. Performance overhead with respect to the period of the self-testing routine (a) and the duration of the self-testing routine (b)

Table 5.3. Area synthesis results of the TMH and STM modules as well as the base tile architecture

	Occupied Slices	**Slice Registers**	**Slice LUTs**
Base tile	2632	6579	8985
Base tile + TMH	2717	6749	9051
Base tile + STM + TMH	2843	6912	9336

5.5.3 Architectural support hardware overhead

The area synthesis results of the TMH module obtained with the Xilinx ISE tool are reported in table 5.3. The area overhead due to the STM and TMH modules is only 8.0% as opposed to 20.7% in the CRR-based technique for M-JPEG described in section 5.2. The STM and TMH modules are clearly scalable as their size depends on neither the number of tiles of the platform nor the number of tasks and channels of the application unlike the CRR-based scheme.

We could not experiment with larger NoC dimensions due to the size of the FPGA at hand. However, we expect the recovery time to be scalable due to the fact that main actors involved in the recovery such as the RM, the new nodes and, if application tasks are mapped optimally, the successor/predecessor tiles are likely to be located close to the faulty node.

In order to employ the proposed recovery technique in hard real-time systems, one can determine the worst case recovery time given a NoC that supports bandwidth reservation. We consider this outside the scope of this work.

5.6 Summary

In this chapter, we have proposed two fault recovery mechanisms that both require modifications at the application, run-time and hardware levels. The envisioned fault-aware run-time environment has been realized for PPN applications running on NORMA-based NoC multiprocessor platforms. The techniques have been evaluated with the M-JPEG encoder and H.264 decoder case studies.

The CRR mechanism is prone to having a significant steady-state performance overhead due to the execution of the self-testing routine at each iteration of a task body. Morever, it requires a more complex TMH module that incurs a large overhead in area, which increases linearly with the number of channels in the application running on the platform.

On the other hand, the RFR mechanism has a much lower steady-state performance overhead due to the fact that self-testing is done less frequently with any desired period. The period has an impact on the amount of error propagation. Moreover it requires a much simpler TMH module that incurs a small overhead. Most importantly, the area of the TMH in RFR mechanism is independent of the application size.

For the considered multimedia use cases, both techniques had short recovery times which allow the system to react to faults without a substantial impact on the user experience.

Chapter 6

Application-level Self-adaptation for Quality Management

Quality management is an activity aiming to make a system deliver the expected quality when performing its actions. In our context, since we are dealing with throughput-oriented systems, the quality refers to the throughput of the system. This chapter focuses on realizing quality management via self-adaptation of some application level parameters and addresses some of the challenges described in section 1.1.1, namely, adaptation management, adaptation overhead and separation of concerns. The results presented in this chapter have been published partially in [Derin et al., 2012; Derin, Ramankutty, Meloni and Tuveri, 2013; Derin and Ferrante, 2009].

The remaining part of the chapter is organized as follows. Section 6.1 discusses the contributions of this dissertation with respect to the state-of-the-art. Sections 6.2 and 6.3 introduce the two proposed approaches for a self-adaptive framework with implementation details of monitoring, controlling and adaptive tasks. Section 6.4 gives details on how a self-adaptive M-JPEG encoder case study is built using the two frameworks. Sections 6.5 and 6.6 give the results of the case study with a comparison of the two approaches in terms of the steady-state performance overhead and quality of the control.

6.1 Contributions with respect to the state of the art

As highlighted by [Nollet et al., 2010], run-time management techniques can be devised for various aspects such as quality of service, power, temperature, variability and load balancing. With our work in this chapter, we aim at addressing quality management via application adaptation. In fact, this work is comple-

mentary to the fault tolerance support described in the previous chapters and it can be integrated orthogonally into a fault-aware run-time environment. The quality management support may be beneficial to satisfy application goals if the performance degrades beyond an acceptable quality level after a remapping due to a fault. Beside such a benefit, our work is motivated by some shortcomings of the previous work on application-level adaptability and quality management in multimedia systems overviewed in section 2.9.

Dynamic models of computation are intented mainly for applications requiring dynamicity rather than self-adaptation. Therefore MoCs such as P^3N [Zhai et al., 2011] and PSDF [Neuendorffer and Lee, 2004] lack the monitor and controller to make them self-adaptable; however, they can be easily extended as they provide built-in support for consistent reconfiguration of application parameters. We rely on concepts similar to quiescent points and reconfiguration ports as used in P^3N and PSDF for parameter reconfiguration.

Previous QoS control approaches for multimedia applications focus on the control aspect of the problem. In fact, in our work, we also adopt fuzzy control similar to [Grant et al., 1997; Rezaei et al., 2006]. On the down side, control-centric approaches such as [Bridges et al., 2009; Grant et al., 1997; Rezaei et al., 2006] address the adaptation problem in an ad-hoc and application specific manner. As in the case of [Bridges et al., 2009], creating a barrier for synchronizing all system components requires application knowledge to decide the barrier's location and may incur unnecessary blocking of portions of the system leading to a loss of performance during parameter reconfiguration.

In comparison to [Cornbaz et al., 2005, 2007; Jaber et al., 2008], our work differs in two aspects: firstly, we target applications running on MPSoCs in a distributed manner, whereas these techniques consider single threaded applications; secondly, our controller is generic and requires minimal knowledge of application characteristics as compared to these methods, which require exhaustive profiling of each task of the application for all quality levels.

Adaptability in NoC platforms has been demonstrated by [David et al., 2011] for hardware level parameters such as voltage and frequency on the Intel SCC chip and by [Clermidy et al., 2011] for reconfiguring the mode of the 3GPP-LTE application on the Magali chip. [Clermidy et al., 2011] presents an ad-hoc solution only for the reconfiguration problem without incorporating an adaptation control that satisfies a particular goal. Our work presents a framework by which a complete self-adaptive application is realized in a NoC platform.

One of the self-adaptation schemes proposed in this chapter, namely MCA-EI, requires the inter-processor interrupt (IPI) support from the platform. Some emerging NoC-based multi-core architectures provide such support. For exam-

ple, Tilera's Tile64 can deliver interrupts to notify user-space processes of message arrival [Wentzlaff et al., 2007]. This allows it to support both polling and interrupt-based message delivery. Intel's SCC also provides a hardware message passing mechanism that triggers an interrupt on the receiving core, before returning from a *write()* call [Howard et al., 2011]. To the best of our knowledge there is no work that evaluates the impact of the use of interrupts on the overhead related to the self-adaptation of the system.

The self-adaptation mechanism proposed in this work relies on monitoring, controlling and adaptation capabilities. For the monitoring and adaptation support, despite the fact that some general mechanisms such as monitoring and adaptive functions are used, the methods are based on some advantages that come with the PPN computation model. In this work, we deal specifically with throughput monitoring and parametric adaptations. The PPN model facilitates implementation of such monitoring and adaptation capabilities. For the former, since PPN is composed of computational blocks and their explicit communication with tokens over channels, monitoring the throughput (e.g., the rate at which tokens are produced as well as the bit-rate on a channel) can be achieved in an application independent manner. For the latter, the reconfiguration of the application to work with a new value of an application parameter requires that the relevant parts of the application are updated consistently. Consistency implies that a token is processed by tasks throughout the application pipeline with the right parameter value. PPN helps achieving that in two ways. First, it allows identifying the execution point at which the parameter value can be changed. Secondly, it allows synchronizing the updating of tasks via blocking channels. These properties of the PPN model address the *separation of concerns* challenge by relieving the programmer of such duties.

On the other hand, the fuzzy control approach is not specific to PPN and can be used for controlling any self-adaptive system. Unlike the widely practiced periodic monitoring and control in conventional systems, this approach involves monitoring in an event-based manner (e.g., at the end of processing of a data unit). Such an approach is suited better for networked systems as it is less sensitive to possible delays in the network and incurs less overhead on the amount of data transferred on the network.

Our contributions in this field are

- a self-adaptation framework based on a monitor-controller-adaptor loop that interacts with the application via blocking channels,

- a self-adaptation framework based on a monitor-controller-adaptor loop

that interacts with the application via interrupting messages, which requires a platform that supports inter-processor interrupts,

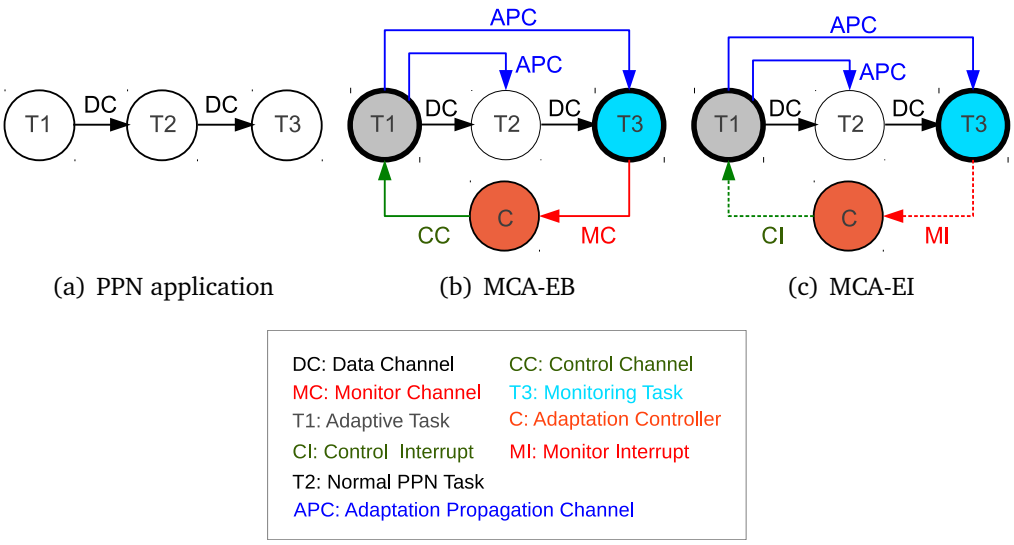
- investigation of fuzzy control as a generic adaptation management mechanism for self-adaptive systems,
- evaluation of the proposed approaches with a case study and their comparison in terms of steady-state performance overhead and quality of control.

6.2 MCA-EB: Self-adaptation with blocking channels

This section presents our framework to build self-adaptive component based applications by incorporating a distributed monitor-controller-adapter (MCA) mechanism in the PPN application pipeline (as proposed in [Derin and Ferrante, 2009]). The application is augmented by special tasks that monitor, control, adapt, and while doing so, communicate events with the blocking channel semantics also used by application tasks. Therefore, we name this scheme as *event-based MCA using blocking channels* (MCA-EB). Monitoring involves measurements of various parameters to check whether the system meets the assigned goals. The controller is capable of driving adaptations when goals are not met, whereas adapters are in charge of performing adaptations. In case of PPN applications running on MPSoCs, various tasks of the application will be mapped onto different tiles of the platform. Hence it is quite possible that the parameter to be monitored is present in one tile, whereas the task to be adapted may exist on a different tile. This forces the monitor, controller and adapters to be implemented on different tiles in a distributed manner. For example in the case of a video encoder application, bit-rate monitoring should be performed on the tile where the sink task is present whereas the frame-size adapter logic has to be present on the tile where the source task is located. Our framework represents a self-adaptive application in terms of the following entities: adaptive tasks implementing adapter functions, monitoring tasks calling monitoring functions, adaptation controller(s) and adaptation propagation channels that augment the original task graph. Figure 6.1(b) depicts a simple PPN application and its self-adaptive version based on our framework.

6.2.1 Adaptive task

In order to implement application specific adaptations, each task should expose its adaptation space (set of adaptable parameters) to the external world. Adap-



(a) PPN application (b) MCA-EB (c) MCA-EI

(d) Legend

Figure 6.1. Self-adaptation approaches for PPN applications on NoC.

tive tasks will have *control channels* and multiple optional *adaptation propagation channels* in addition to nominal input/output data channels. Control channels carry the control commands from the controller to adaptive tasks whereas adaptation propagation channels carry new parameter values from adaptive tasks to other tasks which require these updated values. For example in case of an adaptive source task (which supports frame-size adaptation) in a video encoding application, the control channel will carry the frame-size control command from the controller, whereas the adaptation propagation channels will carry the new frame-size to any other relevant tasks. The frequency at which these channels will be read/written by the task depends on the application as well as the granularity required for the control. In order to perform the adaptation, the task should read the control command from the control channel and call the adapter functions, with control command as the argument. It should also send the modified values of the adapted parameter to other tasks which need these updated parameters. Figure 6.2 shows the modifications required (shown in blue) to transform a PPN task into an adaptive PPN task.

Adapter functions

Adapter functions perform the actions needed to perform the adaptations. The implementation of adapter functions are parameter dependent. An adapter func-

```
adaptiveTask()
{
  for(i=0; i<M; i++) {

    read(CTRL_CH, &ctrlSignal);
    adaptParam(ctrlSignal);
    write(ADAPT_PROP_CH, newParam);

    for(j=0; j<N ; j++) {
      read(DATA_IN_CH, &inData)
      outData = process(inData);
      write(DATA_OUT_CH, outData);
    }
  }
}
```

Figure 6.2. An adaptive task

tion takes care of adapting a parameter by accepting a control command as its argument. The control command can take one of the following values: a) −2: modify the adaptable parameter so as to aggressively reduce the monitored parameter b) −1: modify the adaptable parameter so as to mildly reduce the monitored parameter c) 0: maintain same value for the parameter d) +1: modify the adaptable parameter so as to mildly increase the monitored parameter e) +2: modify the adaptable parameter so as to aggressively increase the monitored parameter. The adapter functions need to be implemented by the application programmer with appropriate interpretation of the mild/aggressive changes to the parameter.

6.2.2 Monitoring task

Monitoring refers to the measurement of a parameter in the system that is of interest. The accuracy and timing of these measurements are critical, since it impacts the overall quality of adaptation. A normal PPN task is converted to a monitoring task by calling monitoring functions provided by the framework. Figure 6.3 shows a simple monitoring task obtained by modifying a typical PPN task by adding calls to the monitoring functions (shown in blue). Our framework supports two types of throughput monitoring: bit-rate and token-rate. The granularity of monitoring is application dependent and it is the application programmers responsibility to insert calls to the monitoring functions at an appropriate place in the code. Furthermore, the framework assumes support from the platform to measure the current time. Monitoring task should also send the monitored

```
monitoringTask()
{
  for(i=0; i<M; i++) {

      int dataCounter = 0;
      for(j=0; j<N ; j++) {
         read(DATA_IN_CH, &inData);
          dataCounter += sizeof(inData);
          outData = process(inData);
          write(DATA_OUT_CH, outData);
      }

      timeStamp t = getCurrentTime();
      alignSlidingWindow(dataCounter, t);
      br = calculateBitrate();
      tr = calculateTokenrate();
      write(MONITOR_CH_BR, br);
      write(MONITOR_CH_TR, tr);
  }
}
```

Figure 6.3. A monitoring task

parameter values to the adaptation controller using monitor channels.

Monitoring functions

Following are the monitoring functions provided:

alignSlidingWindow: We propose sliding-window monitoring, which is triggered by a call to the alignSlidingWindow function. Sliding window method is deployed to find the average of the most recent few instantaneous values of a monitored parameter. It is realized using two circular arrays of size equal to *monitor-width*, which is configurable in the implementation. These arrays are used to hold the values of the two arguments of this function, namely, the data counter and the timestamp. When alignSlidingWindow is called (with the newly captured data counter value and its timestamp as arguments), the windows are adjusted such that the arrays contain the most recently monitored values.

calculateTokenRate: This function calculates the number of monitoring actions performed per unit time by the monitoring task. It is calculated using the number of entries in the sliding window and the difference in timestamps between the latest and oldest entries. It can be used to measure the throughput in terms of the token (or stream unit) rate produced or consumed by the monitoring task.

calculateBitRate: This function calculates the throughput of the generated or

consumed data in terms of its bit-rate. It is calculated by dividing the sum of all entries in the monitoring window by the difference in timestamps between the latest and oldest entries.

The sliding window enables computing the moving average of the monitored values. It smooths out short-lived fluctuations and emphasizes long-term trends. The size of the sliding window can be specified by the application programmer using the monitor-width parameter. This parameter decides on the sensitivity of the control mechanism (i.e., how fast the variations in the monitored variables are perceived). If the monitor-width is too large the sensitivity will be low, that is, the effect of a particular adaptation strategy will be reflected in the average value only after many values got generated under that strategy. On the other hand, a very small monitor window helps in detecting changes in the parameter very fast. However, this may cause large ripples in the output since any adaptation strategy needs some settling time before its effects are visible. Hence it is important to choose a monitor-width value that leads to a good quality of adaptation. This can be done by means of a DSE phase as will be done in section 6.5.

6.2.3 Controller

The most important entity of any adaptation scheme is the controller, because it takes decisions to steer the monitored parameters towards their target values. The correctness and speed of the decisions taken by the controller influence the effectiveness of the adaptation mechanism. Hence controller is the most critical entity in the design of self-adaptive systems. In order to free the application developer from self-adaptivity concerns, our framework provides a generic *fuzzy logic* [Zadeh, 1965] based adaptation controller that can be associated with a goal given an adaptable parameter that has an effect upon that goal.

Fuzzy logic is a form of multi-valued logic that deals with reasoning in an approximate way rather than precise. It is derived from the fuzzy set theory which is based on the understanding that, every fact is present or not up to a certain degree. Fuzzy control represents a formal methodology for presentation, manipulation and implementation of human heuristic knowledge about how certain processes should be controlled by using a simple, rule-based "*if X and Y then Z*" approach rather than attempting to model a system mathematically. For example, instead of dealing with temperature control in precise terms, fuzzy controller uses linguistic terms such as "*if (process is too cool) and (process is getting colder) then (heat the process)*" or "*if (process is too hot) and (process is heating) then (cool the process quickly)*". These terms are imprecise yet very descriptive of what must actually happen. We chose to use fuzzy logic control since the math-

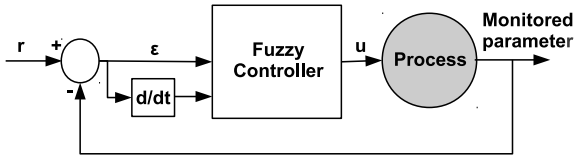


Figure 6.4. A simple fuzzy control based system

ematical models of most application processes are unknown and would be very difficult to build, yet it can easily be described linguistically such as - if process is very hot and the temperature is increasing it is clear that the process has to be cooled quickly.

Figure 6.4 depicts a simple fuzzy logic-controlled system. Here some parameter of interest within the system is monitored. The error signal (ϵ) is the difference between the reference value (r) of the parameter and its monitored value. The fuzzy control logic takes this error signal and its rate of change as inputs and generates the control signal (u) as the output, which will be fed to the adapter logic.

In the case of multiple goals for which the adapted parameters do not have overlapping and conflicting effects on the monitored variables, we can associate a separate fuzzy controller for each specified goal of the system. This also allows controller tasks to be placed in tiles which are at optimum distances from the corresponding adaptive and monitoring tasks, hence reducing the latency and the amount of network traffic introduced by control data. The frequency at which the controller should be run is application dependent. For example in case of a frame-rate control in a video encoder, the algorithm can be run once for every N video frames where N can be chosen by a DSE phase as will be done in section 6.5.

Our design of the fuzzy controller is based on the following parameters.

Error (ϵ): The difference between the monitored value of a parameter and its target value.

Delta Error ($\Delta\epsilon$): The difference between current error and previous error.

Control Settling Width: The duration for which the controller should wait for a control decision to take its effect on the monitored parameter before taking the next decision. In other words, settling width represents the duration between two consecutive control decisions. For example, in the case of frame-rate control, the settling width can be represented in terms of the number of frames between two consecutive control decisions.

Error Threshold Low and *Error Threshold High*: Threshold values divide the error axis into distinct intervals (i.e., error ranges). The decision taken by the controller depends on which interval in the error axis the current error value

Table 6.1. Error ranges for the fuzzy controller

Error Range	**Range Name**
(Error Threshold High) $< \epsilon$	positive huge
(Error Threshold Low) $< \epsilon \leq$ (Error Threshold High)	positive large
$0 \leq \epsilon \leq$ (Error Threshold Low)	positive small
-(Error Threshold Low) $\leq \epsilon < 0$	negative small
-(Error Threshold High) $\leq \epsilon <$ -(Error Threshold Low)	negative large
$\epsilon <$ -(Error Threshold High)	negative huge

Table 6.2. Delta-error ranges for the fuzzy controller

Delta-error Range	**Range Name**
(Delta Error Threshold) $< \Delta\epsilon$	positive large
$0 \leq \Delta\epsilon \leq$ (Delta Error Threshold)	positive small
-(Delta Error Threshold) $\leq \Delta\epsilon < 0$	negative small
$\Delta\epsilon <$ -(Delta Error Threshold)	negative large

belongs to.

Delta Error Threshold: Similar to the error thresholds, delta-error threshold divides the delta-error axis into sub-intervals (i.e., delta-error ranges). The interval in which delta-error falls also influences the decision of the controller.

Error and delta-error values are assigned a range name depending on which interval the value of error/delta-error falls. Tables 6.1 and 6.2 give all the possible ranges and the corresponding range names for errors and delta-errors, respectively.

Our controller implements five discrete levels of control as detailed in table 6.3. For example, to reduce the monitored parameter aggressively, controller generates -2 as the control command. Similarly a $+1$ at the controller output seeks for mild increase in the parameter. The interpretation of these discrete outputs are parameter dependent and has to be done by the adapter functions.

The decision making algorithm of the controller is summarized as follows:

- If error range is *positive huge* then control command is -2 (i.e. if the current value of the parameter is very much greater than the target value then seek to decrease it aggressively).

- If error range is *positive large* and delta-error range is *negative large* then control command is 0 (i.e. if the current value of the parameter is greater than the target value and the error is decreasing at a very fast pace then

Table 6.3. Control levels and their meanings

Control levels	Meaning
-2	Aggressively reduce the monitored parameter
-1	Mildly reduce the monitored parameter
0	Maintain same value for the monitored parameter
$+1$	Mildly increase the monitored parameter
$+2$	Aggressively increase the monitored parameter

seek to maintain previous situation. This means that the decision taken at the previous step was correct, so do not change anything).

- If error range is *positive large* and delta-error range is not *negative large* then control command is -1 (i.e. if the current value of the parameter is greater than the target value and the error is not decreasing at a very fast pace then reduce the parameter mildly. This means that the decision taken at the previous step was not effective enough and further reduction of parameter value is needed).

- If error range is *positive small* and delta-error range is *positive large* then control command is -1 (i.e. if the current value of the parameter is slightly greater than the target value and the error is increasing at a very fast pace then reduce the parameter mildly. This means that even though the error is within the tolerance band it is deviating in the positive direction very fast. So try reducing the parameter value mildly).

- If error range is *positive small* and delta-error range is not *positive large* then control command is 0 (i.e. if the current value of the parameter is slightly greater than the target value and the error is not increasing at a very fast pace then seek to maintain previous situation. This means that the error is smoothly maintaining its value within the tolerance limits, so no action needed).

- If error range is *negative small* and delta-error range is *negative large* then control command is $+1$ (i.e. if the current value of the parameter is slightly lesser than the target value and the error is decreasing at an abrupt pace then increase the parameter mildly. This means that even-though error is within the tolerance band it is deviating in the negative direction very fast. So try increasing the parameter value mildly).

- If error range is *negative small* and delta-error range is not *negative large* then control command is 0 (i.e. if the current value of the parameter is slightly lesser than the target value and the error is not decreasing fast then nothing needs to be changed. This means that error is smoothly maintaining its value within the tolerance limits, so no action needed).
- If error range is *negative large* and delta-error range is *positive large* then control command is 0 (i.e. if the current value of the parameter is much smaller than the target value and the error is increasing at a very fast pace then seek to maintain previous situation. This means that the decision taken at the previous step was correct, so no action required).
- If error range is *negative large* and delta-error range is not *positive large* then control command is +1 (i.e. if the current value of the parameter is much smaller than the target value and the error is not increasing at a very fast pace then seek to increase the parameter mildly. This means that the decision taken at the previous step was not sufficient and further increase of parameter is needed).
- If error range is *negative huge* then control command is +2 (i.e. if the current value of the parameter is very much smaller than the target value then seek to increase it aggressively).

Table 6.4 captures the behavior of the algorithm for all possible situations.

The operation of the controller can be summarized as below. For every new received value of the monitored parameter, the controller decides whether to take a new control decision depending on the settling-width. If this received value has to be ignored for a parameter then the corresponding adaptive task will be asked to maintain its previous situation (by sending 0 as the control command). On the other hand, if the received value has to be considered for a parameter then following actions are performed. Error and delta-error for that parameter are calculated first. Then control algorithm will be run using these values to decide the control command. The generated command will be communicated to the respective adaptive task through the control channel.

6.3 MCA-EI: Self-adaptation using inter-processor interrupts

In this section, we present another approach, *event-based MCA using inter-processor interrupts* (MCA-EI), for implementing self-adaptive PPN applications on NoC-

Table 6.4. Adaptation control algorithm

		Delta-error			
		positive large	positive small	negative small	negative large
Error	positive huge	-2	-2	-2	-2
	positive large	-1	-1	-1	0
	positive small	-1	0	0	0
	negative small	0	0	0	1
	negative large	0	1	1	1
	negative huge	2	2	2	2

based MPSoCs. Similar to MCA-EB, MCA-EI introduces an MCA feedback loop into the application pipeline. The monitor (equivalent to sensors) measures various parameters to check whether the application meets its goals. The controller takes decisions so as to steer the system towards the goal, whereas adapters (similar to actuators) are in charge of actually performing adaptations. Since processes are generally mapped to different resources on MPSoCs, it is quite possible that the parameter to be monitored is present on one tile, whereas the task to be adapted may exist on a different tile. This forces the monitor, controller and adapters to be implemented on different tiles in a distributed manner. Both MCA-EB and MCA-EI incorporate a generic fuzzy logic based adaptation controller and implement similar monitoring and adaptation techniques. Similarly both use event-based control; which means the adaptation control is triggered upon the occurrence of specific events in the system. However they differ based on how the MCA mechanism interacts with the application. MCA-EB uses *blocking-channels* to this end, whereas MCA-EI is based on *inter-processor interrupts*.

Similar to MCA-EB scheme, MCA-EI also represents a self-adaptive application in terms of the following entities: monitoring tasks calling monitoring functions, adaptive tasks implementing adapter functions, adaptation controller(s) and adaptation propagation channels alongside the original task graph. A simple self-adaptive application pipeline built using MCA-EI framework is shown in figure 6.1(c). The design and usage of monitoring functions, adapter functions and fuzzy adaptation control algorithm are the same for both MCA-EI and MCA-EB. Hence only the main differences of MCA-EI in comparison with the MCA-EB are presented here.

Figure 6.5 depicts the pseudo-code representing a monitoring task in MCA-EI scheme. The modification performed on a normal PPN task to convert it to a monitoring task (by adding calls to monitoring functions) is colored in blue. In this example the task is equipped with throughput (in terms of bit-rate) monitor-

```
monitoringTask()
{
  for(i=0; i<M; i++) {

      int dataCounter = 0;
      for(j=0; j<N ; j++) {
         read(DATA_IN_CH, &inData);
          dataCounter += sizeof(inData);
          outData = process(inData);
          write(DATA_OUT_CH, outData);
      }

      timeStamp t = getCurrentTime();
      alignSlidingWindow(dataCounter, t);
      adaptIntr.Value = calculateBitrate();
      adaptIntr.Type = MONITOR_INTR;
      sendInterrupt(BR_CTRL_TILE, adaptIntr);

  }
}
```

Figure 6.5. A monitoring task in MCA-EI scheme.

ing capabilities. The difference to be noted compared to MCA-EB is that, instead of sending the monitored parameter value over blocking channel it is sent as an interrupting message to the NoC tile where the controller is run.

As compared to MCA-EB (where the controller is a separate task), the control algorithm is run as part of the interrupt handler in MCA-EI. As shown in the pseudo-code of the controller given in figure 6.6, the interrupt handler handles two kinds of interrupts; a) monitor interrupts (MI) - from monitoring task's tile to adaptation controller's tile, b) control interrupts (CI) - from adaptation controller's tile to adaptive task's tile. Upon receiving an MI, the interrupt handler runs the fuzzy control algorithm with the received monitored parameter value as the argument to generate the control command. Subsequently it interrupts the adaptive task's tile (using CI) to send the control command. On the other hand, CI is handled as follows; first the received control command will be cached so that it can be processed by the adaptive task later, then a flag is set to inform the adaptive task that a control interrupt had occurred.

An adaptive task in MCA-EI is shown in figure 6.7. It checks for any previous interrupts from the controller tile at a fixed location in the task body. In case of any previous interrupts the adaptive function will be invoked (with the cached value of control command) to perform the required actions. Further it also sends the modified values of the adapted parameter to other tasks (using APC) which need these updated parameters. In PPN model, processes can receive external input only via blocking FIFO channels. Checking the pending controller inter-

```
adaptIntrHandler(adaptIntr)
{
  switch(adaptIntr.Type) {

    case MONITOR_INTR:
      adaptIntr.Value = fuzzyCtrl(adaptIntr.Value);
      adaptIntr.Type = CONTROL_INTR
      sendInterrupt(ADAPT_TASK_TILE, adaptIntr);
    break;

    case CONTROL_INTR:
      cacheCtrlParam(adaptIntr.Value);
      CtrlIntrPending = TRUE;
    break;
  }
}
```

Figure 6.6. Adaptation interrupt handler in MCA-EI scheme.

rupt flag is a non-blocking operation, thus it does not affect the liveness of the application.

The functioning of MCA-EI can be summarized as follows. When the event which triggers the adaptation control is generated, the monitoring task performs monitoring and interrupts the adaptation-controller's tile. The controller tile receives this interrupt and runs the control algorithm as part of the interrupt handler. Subsequently, the controller interrupts the tile of the adaptive task to send the control command. The interrupt handler of the adaptive task tile caches the interrupts received, to be processed by the adaptive task later. When the adaptive task reaches the predefined point of execution, it checks the presence of a cached interrupt and performs the required adaptations if present.

The MCA-EI approach increases the application throughput by using an interrupt mechanism instead of blocking FIFO channels in the MCA feedback loop. MCA-EI scheme has the advantage of not stalling the application pipeline since the adaptive tasks never wait for blocking control commands from the controller. This helps the system to attain higher throughput as compared to the MCA-EB scheme.

In our work, the correctness of the reconfiguration relies on the PPN processes reaching an execution point at which their state does not relate to any computation done using the parameter to be updated. An adaptation command via a control channel or an adaptation propagation channel can only be served at such points. For a general KPN application, the identification of the adaptation execution points and adaptation propagation channels is to be done manually by the application programmer. It should be done carefully to guarantee a cor-

```
adaptiveTask()
{
  for(i=0; i<M; i++) {

    If(ctrlIntrPending) {
      getCachedCtrlCmd(&ctrlCmd);
      adaptParam(ctrlCmd);
      write(ADAPT_PROP_CH, newParam);
    }

    for(j=0; j<N ; j++) {
      read(DATA_IN_CH, &inData)
      outData = process(inData);
      write(DATA_OUT_CH, outData);
    }
  }
}
```

Figure 6.7. An adaptive task in MCA-EI scheme.

rect parameter reconfiguration. Otherwise, the functional correctness may be compromised and also deadlocks may be introduced. In the specific case of PPN [Verdoolaege, 2010], the state of processes reduce to the iterator values at the top of the for-loops of each process. Therefore such locations, as shown in figure 6.7, are ideal points for adapting PPN processes.

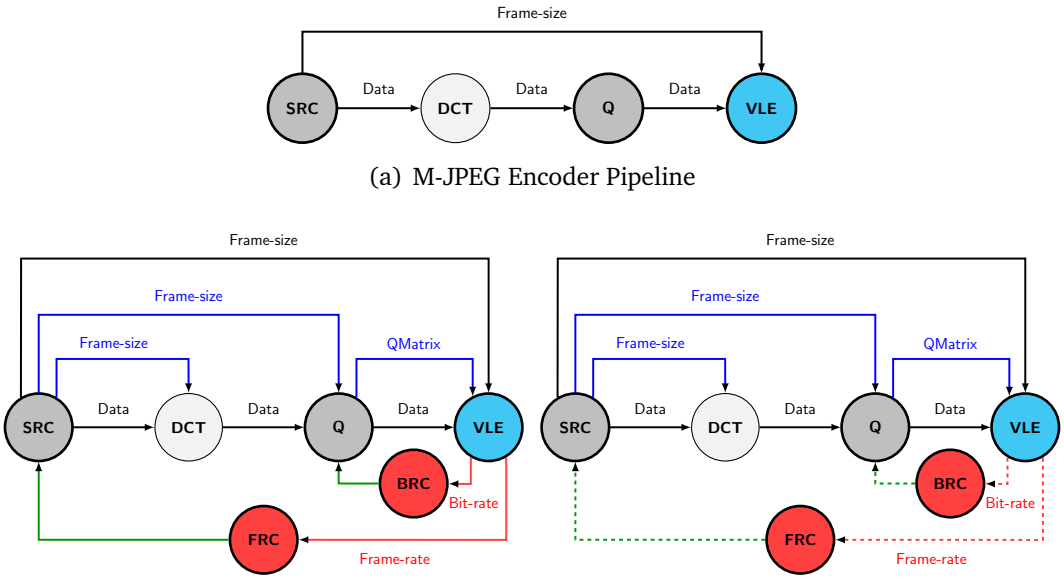
6.4 Case study: Motion JPEG

This section presents Motion JPEG (M-JPEG) [Lieverse et al., 2001], a popular video compression standard, as a case study to demonstrate our framework. This algorithm is selected because its processes are coarse-grained with high computation/communication ratio, a characteristic of an application suited for NoC based MPSoCs. A typical M-JPEG encoder pipeline is shown in figure 6.8(a), where all the components can be modeled as PPN tasks.

Video Source (SRC): This component captures the input video frame-by-frame and feeds it to the succeeding components in the pipeline one block (8×8 pixels) at a time.

Discrete cosine transform (DCT): This component performs discrete cosine transform on each video block received from the SRC component and sends it to the Quantizer for further processing. DCT is widely used for multimedia compression algorithms such as MP3, JPEG and MPEG, where high frequency components of small amplitude can be discarded without compromising quality.

Quantization (Q): Quantization refers to reducing the amplitude of a signal



(a) M-JPEG Encoder Pipeline

(b) MCA-EB

(c) MCA-EI

Figure 6.8. Self-adaptive M-JPEG encoders in MCA-EB and MCA-EI (refer to the legend of figure 6.1(d))

to achieve compression. In M-JPEG, an 8×8 matrix of coefficients (QMatrix) is used for this purpose and the resultant data is rounded off to the nearest integer. The Quantizer also performs a 2D to 1D conversion of the quantized blocks by doing a zig-zag scan.

Variable Length Encoding (VLE): VLE is the last stage of M-JPEG pipeline, where entropy (Huffman) encoding is done on the received video blocks. VLE also acts as the sink component, generating the final M-JPEG stream by inserting headers/markers to indicate the start/end of each frame.

6.4.1 Self-adaptive M-JPEG with MCA-EB

In this section, we present the implementation of the self-adaptive M-JPEG encoder on our NoC-based platform using our MCA framework. This implementation supports autonomous control of bit-rate (BR) and frame-rate (FR) to match the target values set by the user. Bit-rate adaptation is achieved by controlling the quality of encoding (by scaling the QMatrix accordingly), whereas frame-size scaling is used to control the frame-rate. The modifications done on the M-JPEG pipeline to make it self-adaptive are shown in figure 6.8(b) and are as follows:

Table 6.5. Settling widths and error thresholds for controllers (Fraction parameters shown in italic).

Controller parameter	Value
Settling width (FR)	*Settling width factor (FR)* × monitor-width
Error threshold low (FR)	*Error threshold low factor (FR)* × target FR
Error threshold high (FR)	*Error threshold high factor (FR)* × target FR
Delta error threshold (FR)	*Delta error threshold factor (FR)* × target FR
Settling width (BR)	*Settling width factor (BR)* × monitor-width
Error threshold low (BR)	*Error threshold low factor (BR)* × target BR
Error threshold high (BR)	*Error threshold high factor (BR)* × target BR
Delta error threshold (BR)	*Delta error threshold factor (BR)* × target BR

Monitoring VLE

The VLE task is equipped with monitoring capabilities (for bit-rate and frame-rate) by adding calls to the monitoring functions. Monitoring is done at the frame-level, hence these function calls are made after the task has accumulated all the blocks corresponding to one frame. Timestamp of a frame is measured by reading the hardware timer register of the NoC platform. Every time a new frame is generated, *alignMonitorWindow()* function is called with the frame-size and timestamp arguments. Average values of the bit-rate and frame-rate are obtained by calling *calculateBitRate()* and *calculateTokenRate()* functions, respectively.

Controllers

We decided to implement two independent controllers, one for frame-rate (FRC) and the other for bit-rate (BRC). The design principles are as detailed in the previous section. In our implementation the settling-widths are specified as a fraction of the monitor-width whereas the threshold values of error and delta-error signals are taken as a percentage of the target parameter values. These fraction parameters are exposed so that they can be fine tuned. Calculation of settling-widths and threshold values used in our implementation are shown in table 6.5. For every newly generated frame, the controllers receive the monitored values of bit-rate and frame-rate. Depending on the settling-width, they decide whether to take a new control decision. If a new control decision is needed, the fuzzy control algorithm is run using the error and delta-error values for the input. The generated control-command for bit-rate is sent to the adaptive Quantizer task, whereas the frame-rate control-command is sent to the adaptive Source task.

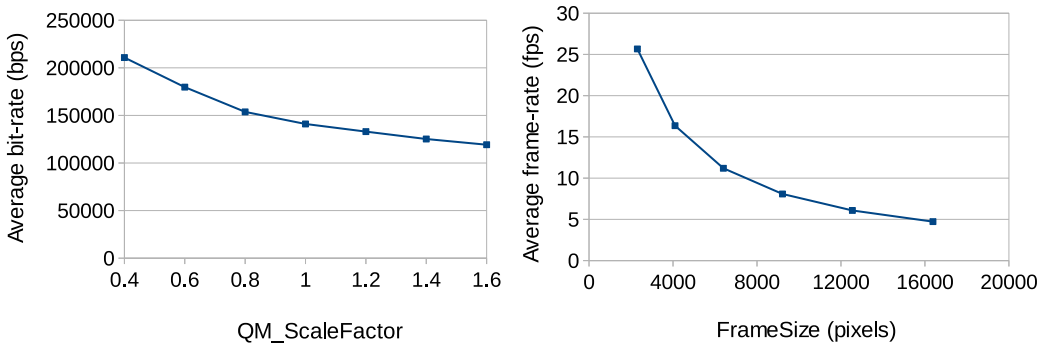
Adaptive Quantizer

The quantization of the data has a direct impact on the generated bit-rate of the encoder. The output bit-rate can be adapted to the required level by scaling the QMatrix. For example, when the quantization coefficients are small, the output of the quantizer has more nonzero values and hence the VLE component will produce more bits per frame. On the other hand, when the input data is quantized using large quantization coefficients, fewer bits will be generated per frame. Figure 6.9(a) shows the output bit-rates for various scaling factors of QMatrix in case of a slow, 128×128 pixel video.

To make the quantizer adaptive, *adaptBitrate()* function is implemented, which takes the control command from the bit-rate controller as input. The implemented bit-rate adapter logic supports two levels of scaling for the QMatrix - aggressive and mild. The algorithm maintains three parameters (configurable by the user) namely - QuantScaleCoeff, AggrQScaleFactor and MildQScaleFactor to perform the adaptations.

QuantScaleCoeff (Quantization scaling coefficient): The coefficient by which all QMatrix coefficients will be multiplied to produce its scaled version.
AggrQScaleFactor (Aggressive Quantization scaling factor): The constant by which previous value of QuantScaleCoeff will be multiplied/divided to obtain its current value in case of aggressive scaling.
MildQScaleFactor (Mild Quantization scaling factor): The constant by which the previous value of QuantScaleCoeff will be multiplied/divided to obtain its current value in case of mild scaling.

The bit-rate adapter works as follows. Before reading the data for a new frame, the quantizer task reads the bit-rate control command from the controller and calls adaptBitrate() function with this value. If the decision by the controller is to aggressively decrease the bit-rate, the current value of the *QuantScaleCoeff* will be multiplied by AggrQScaleFactor to obtain its new value. On the other hand, if the adapter is asked to mildly increase the bit-rate, previous value of *QuantScaleCoeff* will be divided by *MildQScaleFactor* to get its new value. The value of *QuantScaleCoeff* will be left unchanged to keep the bit-rate at the current level. Once the new value for the *QuantScaleCoeff* is decided, the scaled version of the QMatrix is calculated by multiplying all its elements by this new *QuantScaleCoeff*. For all the blocks of the frame, this scaled version of the QMatrix will be used. The quantizer task also sends the newly generated QMatrix to the VLE through a dedicated adaptation propagation channel so that it can be inserted in the frame header of the generated frame.



(a) Bit-rate vs. QMatrix scaling (b) Frame-rate vs. frame-size scaling

Figure 6.9. Impact of adaptation parameters on goal metrics

Adaptive Source

The output frame-rate is decided by how fast the encoder can complete the processing of one frame. Since the processing time for a frame is proportional to the amount of data contained in it, frame-rate can be controlled by scaling the dimensions of the input video. Even though this will produce smaller images at the output, target frame-rate can be easily achieved by using this method. Figure 6.9(b) shows the impact of the frame-size parameter on the output frame-rate.

The source task is made adaptive by providing the adaptFramerate() function, which takes care of scaling the input frame size. The implementation of frame-size scaling logic is based on the following configurable parameters:
CurFrameNumVBlocks: The number of vertical blocks in the current frame.
CurFrameNumHBlocks: The number of horizontal blocks in the current frame.
AggrFsScaleFactor (Aggressive frame-size scaling factor): The constant by which current value of frame-size (number of vertical and horizontal blocks) will be multiplied/divided to obtain its new value in case of aggressive scaling.
MildFsScaleFactor (Mild frame-size scaling factor): The constant by which current value of frame-size (number of vertical and horizontal blocks) will be multiplied/divided to obtain its new value in case of mild scaling.

The algorithm works as follows. Similar to the bit-rate adaptation, the Source task reads the frame-rate control command from the controller and passes this value to the adaptFrameSize() function. If the decision by the controller is to aggressively decrease the frame-rate, the previous values of the *curFrameNumVBlocks* and *curFrameNumHBlocks* will be multiplied by *AggrFsScaleFactor* to obtain their new values. Similarly, if the adapter is asked to mildly increase the frame-rate,

these parameters will be will be divided by *MildFsScaleFactor* to calculate their new values. To keep frame-rate at the current level the number of blocks in the frame will be left unchanged. The frame-rate adapter also sends the new value of the frame-size to DCT, Q and VLE tasks using separate adaptation propagation channels so that they know exactly how many blocks to be processed for the next frame.

Adaptation propagation channels

Some additional channels need to be added to the pipeline to communicate the changes done by the adaptive tasks to other tasks. A channel to send the scaled version of the QMatrix from Quantizer to VLE is added. This is necessary because the QMatrix used for a particular frame needs to be inserted in its header so that the decoder can use the correct value while decoding the frame. Channels to propagate new frame-size values are also added between Source - DCT and Source - Q tasks. Quantizer and DCT should know the frame-size to calculate the number of blocks to be processed for each frame. To send the frame-size values from Source to VLE, we use the existing channel in the original task graph.

6.4.2 Self-adaptive M-JPEG with MCA-EI

We have also implemented the self-adaptive M-JPEG encoders on our 2×2 NoC-based FPGA platform using the MCA-EI approach as shown in figure 6.8(c). Our implementations support autonomous control of bit-rate (BR) and frame-rate (FR) at run-time. Bit-rate adaptation is achieved by controlling the quality of encoding (by scaling the QMatrix accordingly), whereas frame-size scaling is used to control the frame-rate. The implementation of the sliding-window based monitor, fuzzy-logic controller and the bit-rate/frame-rate adapters are same as those in MCA-EB.

6.5 Results for MCA-EB

In this section, we present the results of running our self-adaptive M-JPEG encoder with MCA-EB on the reference platform described in chapter 3. In the experiments presented in this section, we map all M-JPEG tasks as well as the adaptation controllers on the same core (i.e., SRC, DCT, Q, VLE, FRC, BRC $\rightarrow tile_1$).

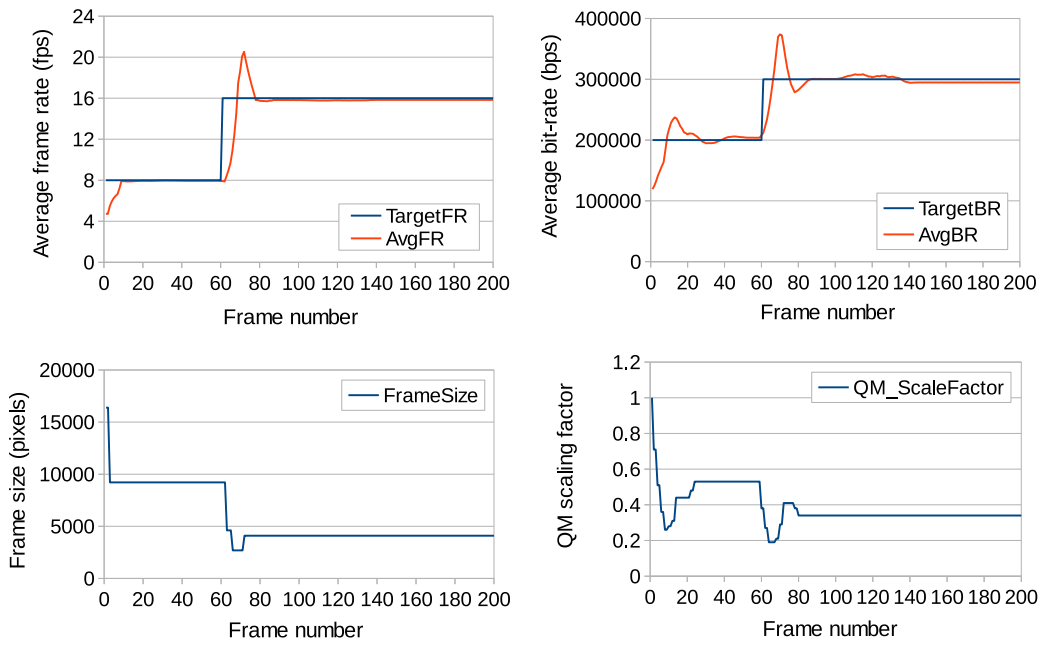
As the criteria to evaluate our approach, we use steady-state adaptation overhead and control quality. Adaptation overhead is measured as the reduction in

Table 6.6. Two step DSE for adaptation control. Selected controller configuration is shown in the last column.

Parameter	**Values**	**After step1**	**After step2**
Monitor width	6, 12, 20	-	12
Settling width factor (FR)	0.1, 0.2	0.2	0.2
Error threshold factor low (FR)	0.1, 0.2	0.2	0.2
Error threshold factor high (FR)	0.2, 0.3	0.3	0.3
Delta-error threshold factor (FR)	0.05	0.05	0.05
Mild frame-size scaling factor	1.1, 1.2	1.1	1.1
Aggressive frame-size scaling factor	1.25, 1.4	1.25	1.25
Settling width factor (BR)	0.1, 0.2	-	0.2
Error threshold low factor (BR)	0.05, 0.1	-	0.05
Error threshold high factor (BR)	0.15, 0.2	-	0.2
Delta-error threshold factor (BR)	0.03	-	0.03
Mild Q scaling factor	1.1, 1.2	-	1.1
Aggressive Q scaling factor	1.4, 1.6	-	1.4

frame-rate and bit-rate, whereas control quality is quantified using rise-time/fall-time (RT/FT) and mean-absolute-error (MAE). To calculate these two metrics for a monitored parameter, the encoder is run for a fixed number of frames of a test video with an initial value of the parameter. Then a target value is set and the system is allowed to adapt. The time taken for the parameter to reach within a tolerance band (±5%) about its target value is the rise/fall time. The absolute error value for the parameter is calculated for all measurements starting from where it reached the tolerance band till the last frame. The mean of these absolute error values gives the MAE value.

As evident from the design of MCA framework, the quality of the adaptation control is influenced by various parameters used inside the monitors, controllers and adapters. In order to achieve smooth and fast adaptation, a careful selection of these parameters is needed. To find such a combination, a DSE is performed. Detailed results of the DSE are reported in [Derin et al., 2012]. In summary, a design space composed of 3072 different controllers has been explored in a greedy manner by evaluating 128 of them in two phases as shown in table 6.6. 95% of the design points have less than 5% bit-rate error, whereas the rise-time of 90% of them are below 8 frames. Similarly, for frame-rate control, 80% of the design points have less than 12% frame-rate error, whereas the rise-time of 84% of them are below 9 frames. This shows the generality of the proposed controller, because even for non-optimal parameter configurations, the system is



(a) Average frame-rate and frame size variations (b) Average bit-rate and QM scaling factor variations

Figure 6.10. Results for initial FR = 8 fps, initial BR = 200000 bps and final FR = 16 fps, final BR = 300000 bps

able to adapt fast while keeping the error within tolerable limits. The selected Pareto point (shown in the last column of table 6.6) obtained from the DSE is used to configure the controller for the experiments in the following sections.

6.5.1 Bit-rate and frame-rate adaptation tests

To demonstrate the effectiveness of our adaptation scheme we conducted various experiments by setting different goals for bit-rate and frame-rate. All these tests are carried out using a 128×128 video (for 200 frames). Figure 6.10 shows the results when the encoder is run with initial BR = 200000 bps, initial FR = 8 fps, final BR = 300000 bps and final FR = 16 fps. The adaptation in terms of quantization scaling coefficient and frame-size is also shown. It can be seen that the scaling coefficient is reduced from its initial value of 1 to a value of 0.5 to meet the initial bit-rate. But after frame 60, its value is further reduced to 0.35 to increase the bit-rate to its final value. Similarly, the frame-size is reduced from its initial value of 16000 pixels to 9000 pixels in order to achieve the initial

frame-rate of 8 fps. But after frame 60, it is further reduced to about 4000 pixels to increase the frame-rate to 16 fps. The rise time and mean absolute errors for this scenario are: rise time (BR) = 6 frames, mean absolute error (BR) = 7684 bits (2.56%), rise time (FR) = 9 frames, mean absolute error (FR) = 0.33 frames (2.06%).

6.5.2 Fast video vs. slow video

Figure 6.11 shows the results of evaluating the framework using slow (i.e., static content) and fast video (i.e., dynamic content) inputs. Figure 6.11(a) characterizes the two videos in terms of the number of bytes generated by the encoder per frame (for 128×128 video), when there is no bit-rate and frame-rate control. From figures 6.11(b) and 6.11(c), it can be seen that for both videos the targets are achieved. Table 6.7 shows the quality metrics in the case of fast and slow input videos.

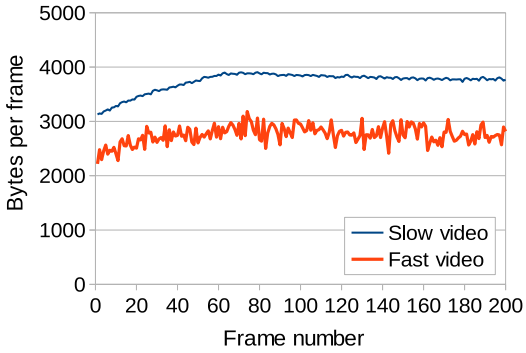
Table 6.7. Comparison of adaptation quality for fast and slow videos

	FR-RT (frames)	**FR-MAE (frames)**	**BR-RT (frames)**	**BR-MAE (bits)**
Slow video	9	0.36 (2.25%)	6	7790 (2.59%)
Fast video	9	0.24 (1.5%)	5	10440 (3.48%)

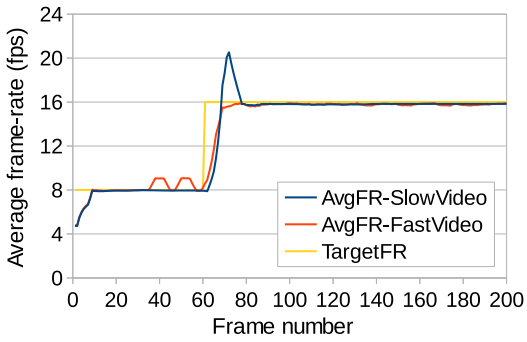
The results reveal that for slow video the bit-rate control converges fast whereas for fast video a lot of ripples are observed at the output, resulting in a higher mean absolute error. In case of frame-rate control, the dynamicity of the input does not have much impact and the frame-size converges to the same value in both cases without ripples.

6.5.3 Cost of adaptation

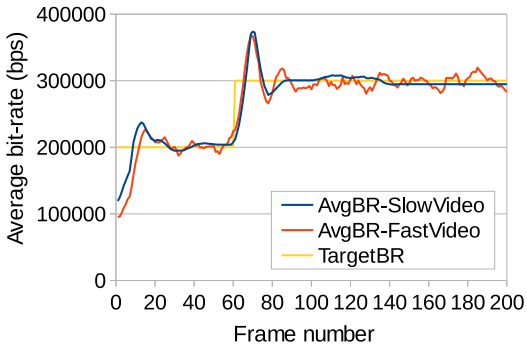
To measure the steady-state overhead due to the introduction of the MCA feedback loop in the application pipeline, the following procedure is used. First, the encoder is run with neither the feedback loop nor the adaptation propagation channels to obtain the average value of frame-rate in absence of the MCA-related changes. The experiment is repeated after introducing the MCA loop and the additional channels to obtain the reduced frame-rate. In this case both bit-rate and frame-rate control is turned off inside the controller, since only the overhead due to the framework needs to be measured. Figure 6.12 depicts the outcome of this test for a 128×128 test video. It is observed that the introduction of the



(a) Number of bytes generated per frame for slow and fast video



(b) Frame-rate control



(c) Bit-rate control

Figure 6.11. Results for bit-rate and frame-rate control of slow and fast videos

framework results in a frame-rate reduction of only 4% and a bit-rate reduction of 3.5%. The reduction in the bit-rate and frame-rate is due to the increase in the inter-arrival time between frames.

The overhead in terms of the additional control data introduced by our MCA mechanism is minimal. For every video frame it sends a total of 72 additional tokens over the network. This includes one token from monitoring task to bit-rate controller, one token from monitoring task to frame-rate controller, one token from bit-rate controller to Quantizer task, one token from frame-rate controller to Source task, 64 tokens from Quantizer to VLE (to send the QMatrix), two tokens from Source to DCT (to send the height and width of the frame) and two tokens from Source to Quantizer (to send the height and width of the frame). This is equivalent to 288 bytes of data - since a token is represented as integer type by the middleware. For a single 128×128 frame the total video data to be sent over the NoC is 49152 bytes. This includes the pixel data sent from Source

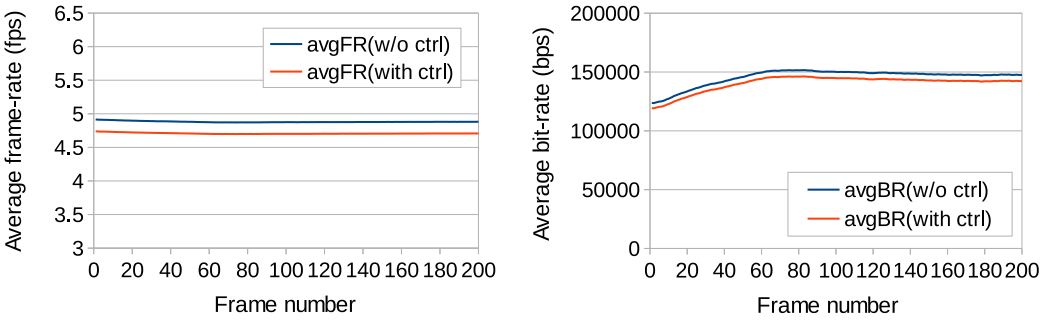


Figure 6.12. Cost of adaptation in terms of reduction in FR (left) and BR (right)

Table 6.8. Comparison of steady-state overheads.

	FR-Overhead (%)	**BR-Overhead (%)**
MCA-EI	0.37	0.37
MCA-EB	6.78	6.78

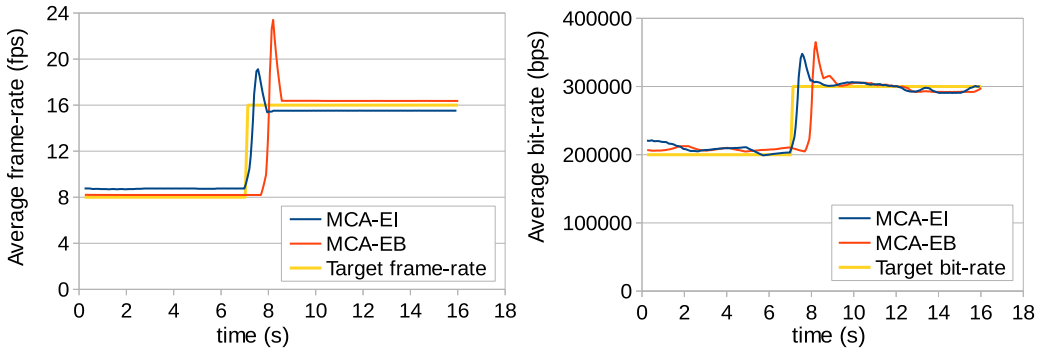
to DCT, DCT to Quantizer and Quantizer to VLE. So the framework introduces approximately 0.5% of additional control data.

6.6 Comparison of MCA-EB and MCA-EI

In this section, the MCA-EI approach is compared against MCA-EB by using the same adaptation scenario. However, differently than the previous section, we map M-JPEG tasks one-to-one on the processing elements as follows: SRC $\rightarrow$ $tile_3$, DCT $\rightarrow$ $tile_1$, Q $\rightarrow$ $tile_2$, VLE $\rightarrow$ $tile_4$. The controllers are mapped together with the heaviest task in order to obtain the worst-case overhead results (i.e., FRC, BRC $\rightarrow$ $tile_1$).

6.6.1 Adaptation overhead

To measure the overhead due to the introduction of the MCA feedback loop in the application pipeline, the following procedure is used. First, the encoder is run without the feedback loop as well as the adaptation propagation channels to obtain the average values of FR and BR without the framework. The experiment is repeated after introducing the MCA loop and the additional channels to obtain the reduced values. In this case both BR and FR control is turned off inside the controller, since only the overhead due to the framework needs to be measured.



(a) Frame-rate adaptation (b) Bit-rate adaptation

Figure 6.13. Results for initial FR = 8 fps, initial BR = 200000 bps and final FR = 16 fps, final BR = 300000 bps.

Table 6.8 compares the steady-state overhead (in terms of bit-rate and frame-rate reduction) for MCA-EI and MCA-EB schemes. It can be seen that the overhead in case of the MCA-EB is much more than MCA-EI because the application pipeline is stalled at the end of every frame. Firstly, the tasks from adaptive tasks till the monitoring task are stalled consecutively until the final frame block is processed by the monitoring task. Only then, the monitoring can be performed and the controllers can provide the control command required to unblock the adaptive tasks. In case of MCA-EI the pipeline is never stalled, yielding higher throughput. Also in MCA-EI, the adapter tiles will not be interrupted by the controller during steady-state; while in MCA-EB, the adaptive tasks have to wait for the control command for every frame even in the steady-state. The throughput reduction in the MCA-EI scheme (0.37%) is due to the execution of the controller inside the interrupt service routine (in our experiments, the controller is mapped on the processor with the heaviest workload).

6.6.2 Control quality

Figure 6.13 shows how the framework adapts (using MCA-EB and MCA-EI) when the encoder is run with initial BR = 200000 bps, initial FR = 8 fps, final BR = 300000 bps and final FR = 16 fps. Here the goals are changed from initial to final at frame 60 and the framework is configured such that the control algorithm is run for every third frame.

Table 6.9 compares the two schemes in terms of rise-time (RT) and mean-absolute-error (MAE) for this experiment. It can be seen that the rise-time for

Table 6.9. Comparison of control quality for MCA-EI and MCA-EB

	RT-BR (s)	RT-FR (s)	MAE-BR (bits)	MAE-FR (frames)
MCA-EI	0.364	0.364	6551 (2.2%)	0.57 (3.6%)
MCA-EB	0.579	0.579	7958 (2.7%)	0.63 (4.0%)

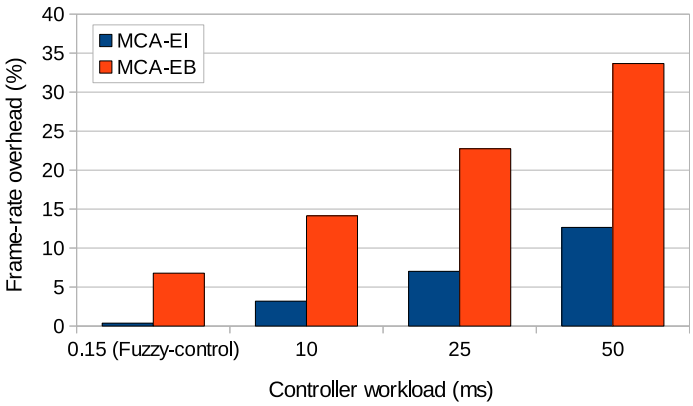


Figure 6.14. Effect of controller workload on adaptation overhead.

the MCA-EB scheme is considerably higher than MCA-EI due to the stalling of the pipeline at the end of each frame. However, the MAE value is within tolerable limits (±5% of final values) for both.

6.6.3 Adaptation overhead vs. Controller workload

Workload of the controller - represented as the calculation time of the control decision, is simulated by introducing delay in the controller code. To obtain the steady-state overhead, the M-JPEG encoder is run with the MCA feedback loop in place but with the bit-rate and frame-rate control turned off. Figure 6.14 shows the variation in the steady-state overhead (as percentage frame-rate reduction) with respect to the controller workload for the MCA-EB and MCA-EI schemes. As obvious, the overhead increases with increasing controller workload. But this increase is far less in interrupt-based scheme as compared to the blocking-channel case. In case of the MCA-EB scheme the entire encoder pipeline is stalled while the control algorithm is run, irrespective of on which tile the controller is located. The throughput reduction in the MCA-EI scheme is due to the controller stealing several cycles from the application task that runs on its tile. These experiments are carried out with the controller running on the processor with the heaviest workload (i.e., the tile of DCT) and shows that the interrupt based approach is

much superior for complex controllers that consume more time. Furthermore, the overhead can be almost eliminated in the MCA-EI scheme by running the controller on a tile where no application task is present. However, such an improvement is not possible for the MCA-EB scheme.

6.7 Summary

In this chapter, firstly, we proposed the MCA-EB approach to implement application level self-adaptation capabilities for PPN applications running on networks-on-chip based MPSoCs. The proposed framework is based on introducing a Monitor-Controller-Adapter mechanism in the application pipeline. Techniques to add monitoring and adaptation capabilities to normal PPN tasks are discussed along with the design of a generic fuzzy logic based adaptation controller. Finally, we presented an adaptive M-JPEG case study on a FPGA based 2×2 NoC platform. Our results show that even if the parameters of the fuzzy control are not tuned optimally, the adaptation convergence is achieved within reasonable time and error limits for most of the designed controllers. Moreover, the steady-state overhead introduced due to the framework is low (6.78%) in terms of frame-rate reduction. Since the controller is a generic one, this framework can be easily integrated to other applications also, requiring minimal modifications to the code.

We also presented the MCA-EI approach aimed at developing low-overhead self-adaptive PPN applications on NoC-based MPSoCs. Compared to the MCA-EB scheme, it makes use of *inter-processor interrupts* to increase the application throughput. Results from the M-JPEG case study show that the MCA-EI scheme outperforms MCA-EB in terms of overhead (0.37% vs. 6.78%) while offering similar or better quality of control. The sensitivity of adaptation overhead to controller workload is also much less in case of MCA-EI. However, MCA-EI requires platform support to send data to remote tiles using interrupting messages over the NoC. This support is implemented in the baseline platform by extending the network interface with a tag decoder.

Chapter 7

Conclusion and Future Work

Keeping up with ever-increasing performance demands and at the same time being curbed by low power constraints, one of the evolution paths for embedded computing systems have been MPSoCs with growing number of processing elements that are interconnected by a NoC. On top of that, a proper computation model and memory model is required in order to unveil the power of scaled computation and communication capabilities. The complexity of underlying platforms and the applications running on them has made some of the conventional design methodologies obselete. A thorough design-time optimization of the system is not possible due to unknown run-time conditions that will be faced by the system. One of such situations is the failure of processing elements due to hard faults, which are becoming a bigger concern with ever-decreasing technology nodes. In this thesis, we make use of the self-adaptation paradigm in order to tackle these challenges within the particular context of KPN-based streaming applications running in NoC-based MPSoCs that adopt the NORMA model. We demonstrate not only that fault tolerance and self-adaptivity can be achieved in embedded platforms, but also it can be done so without incurring large overheads. In particular, this thesis tackled two main problems: (1) fault-aware online task remapping, (2) application-level self-adaptation for quality management. In addressing these problems, we developed techniques which have been realized in the form of either a design tool, a run-time library or a hardware IP core.

Beside introducing the baseline platform and general fault tolerance approach adopted in this thesis, chapter 3 also presented the developed middleware support which enables execution of KPN applications in NoC-based platforms. This was a major requirement in order to realize the self-adaptive run-time environment on the actual platform. Besides satisfying KPN semantics, the middle-

ware allows application tasks to be platform independent with regard to the on-chip communication infrastructure. The middleware is solely based on message passing primitives. The middleware was our initial step towards implementing the application-level self-adaptation and fault-aware online task remapping concepts.

Chapter 4 presented the formulation of the optimal task mapping problem for NoC-based multiprocessors with generic topologies and deterministic routing as an integer linear programming (ILP) problem with the objective of minimizing the communication traffic in the system and the total execution time of the application. We used this to obtain optimal task remappings in presence of faults in processing cores. Several heuristics, which are geared towards run-time use, have been proposed and their results have been compared with respect to the optimal remappings by means of as a synthetic task graph. Our results showed that LNMS heuristics provide a parameterized solution that can accommodate different trade-offs. Comparing the results of the heuristics with the optimal solutions, we have found that they are in average within 7% proximity in terms of degradation distance. Our solution has also been demonstrated with real-life use case applications for several fault scenarios on a mesh-based platform adopting an XY-routing strategy. However the proposed solution can be applied to platforms with any topology and deterministic routing strategy. The results showed that the NMS heuristics are able to find near-optimal remappings with a small run-time penalty, which confirms their adoptability for run-time use in fault recovery mechanisms.

In chapter 4, we also presented an analytical model for estimating the reliability achieved by means of the online task remapping and N-modular redundancy techniques, operating at the application-level and thus abstracting from the underlying hardware architecture (from which fault occurrence-related values are provided). We investigated and evaluated them with respect to the reliability metric (mean-time-to-failure), the overhead in computation (execution time) and communication (amount of data transfer on the network). The analytical model is specific to applications represented as KPNs running on heterogeneous MPSoCs based on NoCs. By presenting a reliability estimator, we allow the possibility to perform reliability-aware design space exploration. A case study validated our technique by showing that the fault-aware online task remapping technique is capable of yielding a two-fold reliability increase (much better when compared to NMR), at the expense of a small overhead which has been assessed in chapter 5.

Chapter 5 presented two techniques that enable fault recovery for PPN applications running on NORMA-based NoC multiprocessor platforms. Both tech-

niques required modifications to the platform at the application, run-time and hardware levels. The techniques have been evaluated on the platform with the M-JPEG encoder and H.264 decoder case studies through fault recovery experiments making use of one of the proposed heuristics. The first technique named CRR relies on the self-testing of the processor at each iteration of a process in order to minimize the error propagation. Therefore it brings a large performance overhead unless the processes have much longer execution times than the self-testing routine. Furthermore, CRR checkpoints many state variables in the register file of the TMH module, which results in a higher area overhead. On the other hand, the second technique named RFR requires an application model in which the processed stream consists of independent stream units. It relies on rolling forward to the next stream unit upon a fault detection by the periodical self-testing of the processors. At the expense of a limited error propagation, the RFR technique incurs a much lower performance and area overhead than CRR due to the reduced register file of the TMH and much smaller number of self-testing invocations.

Chapter 6 introduced two techniques in order to realize application level self-adaptation capabilities for PPN applications running on NORMA-based NoC multiprocessor platforms. Both techniques rely on hooking up the application with a Monitor-Controller-Adapter mechanism that adapts application parameters to meet throughput goals. The first technique, named MCA-EB, interacts with the application through the same semantics as the application tasks (i.e., blocking channels). While the second technique, named MCA-EI, makes use of inter-processor interrupts. Methods that add monitoring and adaptation capabilities to normal PPN tasks are developed along with the design of a generic fuzzy logic based adaptation controller. We experimented with both techniques by realizing an adaptive M-JPEG case study on a FPGA based 2×2 NoC platform. Our results showed that even if the parameters of the fuzzy control are not tuned optimally, the adaptation convergence is achieved within reasonable time and error limits for most of the designed controllers. Since the controller is a generic one, this framework can be easily integrated to other applications also, requiring minimal modifications to the code. Moreover, the steady-state performance overhead introduced due to MCA-EB is low (6.78%) in terms of frame-rate reduction. MCA-EI was proposed with the aim of reducing the steady-state performance overhead even further. Results from the M-JPEG case study showed that the MCA-EI scheme with a steady-state performance overhead of only 0.37% outperforms MCA-EB while offering similar or better quality of control. The sensitivity of adaptation overhead to controller workload is also much less in the case of MCA-EI. However, MCA-EI requires that the platform supports sending

data to remote tiles using interrupting messages over the NoC. This support is implemented in the baseline platform by extending the network interface with a tag decoder.

The work presented in this thesis can serve as a foundation for further research in several directions. We conclude the thesis by listing some of them below.

Incremental improvements to the self-adaptive run-time environment: The proposed techniques for the realization of a fault tolerant and self-adaptive run-time environment can be extended with some incremental improvements. Firstly, the fault tolerance support and application level self-adaptation support can be integrated into the same platform which would allow better flexibility for keeping up application goals via application level adaptations when processors experience permanent failures. Secondly, embedded platforms are designed more often with dynamic voltage and frequency scaling capabilities in order to support a broad spectrum of applications with different power consumption requirements. Remapping problem can be extended to finding the new voltage and frequency values per processing core such that performance degradation is minimized under low power consumption constraints. Thirdly, as embedded platforms are being used more often to run multiple applications at the same time, the mapping and remapping formulation can be extended with the consideration of multi-application scenarios. Although modification of the analytical model for computing the throughput per application is straightforward, remapping heuristics would be more complicated in order to minimize performance degradation for all (or a subset of selected) applications. Lastly, the analytical model for computing throughput used in fault-aware remapping can be extended according to the model recently proposed by Piscitelli and Pimentel [2012] in order to support cyclic task graphs.

Integration of presented techniques into existing design methodologies: In order to address fault tolerance not as a later add-on but rather as a major aspect from the beginning of the design flow, fault tolerance techniques should be embedded into existing design methodologies. A considerable engineering effort is needed to make use of presented techniques in an automated design methodology. Firstly, the design space exploration phase can be extended for fault-awareness by evaluating the lifetime reliability of the system for a given selection of platform components, and by evaluating the degradation of the system under different fault-aware remapping scenarios. Secondly, the fault tolerance related IP cores should be added to the IP repository and instantiated as part of the tile template. Thirdly, the developed run-time support libraries should be included in the run-time support packages. Finally, the process network com-

piler should be extended with a phase that transforms the tasks according to the modified PPN task template.

Lifetime reliability estimation: The lifetime reliability estimation is a promising research topic which is currently limited by the insufficiency of analytical models that capture the causes of faults comprehensively. Our reliability estimation technique presented in section 4.4 lays the basis for further research in system-level reliability-aware design. Present definition of failure in our technique can be improved by considering failure not as the failing of the last fault-free processor, but as the inability to meet the performance goal of the application. This involves carrying out an exhaustive analysis of fault scenarios by varying the order that the processors fail. This task becomes computationally complex and challenging when a heterogeneous platform is considered.

Real-time fault-tolerant scheduling for permanent faults: In our treatment of fault tolerance, we focused on throughput-oriented non-real-time systems and thus elaborated on only remapping and not rescheduling, thanks to the schedulability of KPNs in a data-driven manner. Previous studies that address the rescheduling problem does so either in the case of independent tasks [Krishna, 2014] or transient faults, in which case the rescheduling of tasks are done without considering remapping of the tasks running on the processor that experiences the transient fault [Kang et al., 2014b]. To extend the applicability of our work to real-time systems, techniques are needed for real-time fault-tolerant scheduling in the presence of permanent failures in the processing elements.

Proactive fault management: As the physics of phenomena that cause permanent faults is understood better, the predictability of faults becomes possible. In such a case, the overhead of checkpointing and possible error propagation can be avoided. Identification of relevant parameters and their continuous monitoring pose serious challenges when embedded systems with low resources are considered. If achieved, proactive fault management may become a reality also for embedded systems. Designers have to assess the overhead of both approaches and decide whether to adopt a proactive or, as presented in this thesis, a reactive approach.

Bibliography

Ababei, C. and Katti, R. [2009]. Achieving network on chip fault tolerance by adaptive remapping, *Int. Parallel and Distributed Processing Symposium* **0**: 1–4.

Acquaviva, A., Alimonda, A., Carta, S. and Pittau, M. [2008]. Assessing task migration impact on embedded soft real-time streaming multimedia applications, *EURASIP J. Emb. Sys.* **2008**.

Adapteva Inc., . [2014]. *E16G401 Epiphany 64-core Microprocessor Datasheet*. **URL:** *http://www.adapteva.com*

A High Performance Message Passing Library [n.d.]. **URL:** *http://www.open-mpi.org/*

Al-Ali, R., Hafid, A., Rana, O. and Walker, D. [2004]. An approach for quality of service adaptation in service-oriented grids: Research articles, *Concurr. Comput. : Pract. Exper.* **16**(5): 401–412.

Almeida, G. M., Sassatelli, G., Benoit, P., Saint-Jean, N., Varyani, S., Torres, L. and Robert, M. [2009]. An Adaptive Message Passing MPSoC Framework, *International Journal of Reconfigurable Computing* **2009**: 20.

Amory, A. M., Marcon, C. A. M., Moraes, F. G. and Lubaszewski, M. [2011]. Task mapping on noc-based mpsocs with faulty tiles: Evaluating the energy consumption and the application execution time., *International Symposium on Rapid System Prototyping*, IEEE, pp. 164–170.

Ananthanarayan, S., Garg, S. and Patel, H. D. [2013]. Low cost permanent fault detection using ultra-reduced instruction set co-processors, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp. 933–938.

Ascia, G., Catania, V. and Palesi, M. [2004]. Multi-objective mapping for mesh-based noc architectures, *Int. Conf. on Hardware/Software Codesign and System Synthesis*, pp. 182 – 187.

Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. [2004]. Basic concepts and taxonomy of dependable and secure computing, *Dependable and Secure Computing, IEEE Transactions on* **1**(1): 11–33.

Bacivarov, I., Haid, W., Huang, K. and Thiele, L. [2010]. Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip, *in* S. S. Bhattacharyya, E. F. Deprettere, R. Leupers and J. Takala (eds), *Handbook of Signal Processing Systems*, Springer, pp. 1007—1040.

Bailey, B., Martin, G. and Anderson, T. (eds) [2005]. *Taxonomies for the Development and Verification of Digital Systems*, Springer US.

Balasubramaniam, D., Morrison, R., Mickan, K., Kirby, G., Warboys, B., Robertson, I., Snowdon, B., Greenwood, R. M. and Seet, W. [2004]. Support for feedback and change in self-adaptive systems, *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM, New York, NY, USA, pp. 18–22.

Balevic, A. and Kienhuis, B. [2011]. Kpn2gpu: An approach for discovery and exploitation of fine-grain data parallelism in process networks, *SIGARCH Comput. Archit. News* **39**(4): 66–71.

Bauer, L., Shafique, M., Teufel, D. and Henkel, J. [2007]. A self-adaptive extensible embedded processor, *SASO*, IEEE Computer Society, pp. 344–350.

Benini, L. and De Micheli, G. [2002]. Networks on chips: a new soc paradigm, *Computer* **35**(1): 70 –78.

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

Bhardwaj, K. and Jena, R. [2009]. Energy and bandwidth aware mapping of ips onto regular noc architectures using multi-objective genetic algorithms, *Int. Sym. on System-on-Chip*, pp. 27–31.

Bhattacharyya, S. S., Deprettere, E. F. and Theelen, B. D. [2013]. Dynamic dataflow graphs, *in* S. S. Bhattacharyya, E. F. Deprettere, R. Leupers and

J. Takala (eds), *Handbook of Signal Processing Systems*, Springer New York, pp. 905–944.

Bolchini, C. and Miele, A. [2013]. Reliability-driven system-level synthesis for mixed-critical embedded systems, *Computers, IEEE Transactions on* **62**(12): 2489–2502.

Borkar, S. [2005]. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation, *IEEE Micro* **25**: 10–16.

Borkar, S., Jouppi, N. P. and Stenstrom, P. [2007]. Microprocessors in the era of terascale integration, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, EDA Consortium, San Jose, CA, USA, pp. 237–242.

Bouteiller, A., Hérault, T., Krawezik, G., Lemarinier, P. and Cappello, F. [2006]. Mpich-v project: A multiprotocol automatic fault-tolerant mpi, *Int'l J. High Performance Computing and Applications* pp. 319–333.

Bower, F., Shealy, P., Ozev, S. and Sorin, D. [2004]. Tolerating hard faults in microprocessor array structures, *Dependable Systems and Networks, 2004 International Conference on*, pp. 51–60.

Bridges, P., Hiltunen, M. and Schlichting, R. [2009]. Cholla: A framework for composing and coordinating adaptations in networked systems, *Computers, IEEE Transactions on* **58**(11): 1456–1469.

Bronevetsky, G., Marques, D., Pingali, K., Szwed, P. and Schulz, M. [2004]. Application-level checkpointing for shared memory programs, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, ACM, New York, NY, USA, pp. 235–247.

Brown, G., Cheng, B. H. C., Goldsby, H. and Zhang, J. [2006]. Goal-oriented specification of adaptation requirements engineering in adaptive systems, *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, ACM, New York, NY, USA, pp. 23–29.

Cannella, E., Derin, O., Meloni, P., Tuveri, G. and Stefanov, T. [2012]. Adaptivity support for mpsocs based on process migration in polyhedral process networks, *VLSI Design* **2012**(Article ID 987209): 15 pages. Special issue on Application-Driven Design of Processor, Memory, and Communication Architectures for MPSoCs.

Cannella, E., Derin, O. and Stefanov, T. [2011]. Middleware approaches for adaptivity of kahn process networks on networks-on-chip, *DASIP'11: Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, Tampere, Finland, pp. 1–8.

Carara, E., Mello, A. and Moraes, F. [2007]. Communication models in networks-on-chip, *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, IEEE Computer Society, Washington, DC, USA, pp. 57–60.

Carter, N. P., Naeimi, H. and Gardner, D. S. [2010]. Design techniques for cross-layer resilience, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 1023–1028.

Casas, J. A., Moreno, J. M., Madrenas, J. and Cabestany, J. [2007]. A novel hardware architecture for self-adaptive systems, *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, IEEE Computer Society, Washington, DC, USA, pp. 592–599.

Casavant, T. L., Jon and Kuhl, G. [1988]. A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Transactions on Software Engineering* **14**: 141–154.

Castrillon, J., Schürmans, S., Stulova, A., Sheng, W., Kempf, T., Leupers, R., Ascheid, G. and Meyr, H. [2011]. Component-based waveform development: The nucleus tool flow for efficient and portable software defined radio, *Analog Integr. Circuits Signal Process.* **69**(2-3): 173–190.

Castrillon, J., Velasquez, R., Stulova, A., Sheng, W., Ceng, J., Leupers, R., Ascheid, G. and Meyr, H. [2010]. Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 753–758.

Ceng, J., Castrillon, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T. and Kunieda, H. [2008]. MAPS: an integrated framework for MPSoC application parallelization, *Proceedings of the 45th annual Design Automation Conference*, DAC '08, ACM, New York, NY, USA, pp. 754–759.

Ceponis, J., Kazanavicius, E. and Ceponiene, L. [2008]. Handling multiple failures in process networks, *Information Technology And Control* **37**(1): 19–25.

Chankong, V. and Haimes, Y. [1983]. *Multiobjective Decision Making Theory and Methodology*, North-Holland.

Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P. and Magee, J. [2009]. *Software Engineering for Self-adaptive Systems: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, Springer Berlin Heidelberg.

Chou, C.-L. and Marculescu, R. [2008]. Contention-aware application mapping for network-on-chip communication architectures, *IEEE Int. Conf. on Computer Design*, pp. 164–169.

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

Chou, T. C. K. and Abraham, J. [1983]. Load redistribution under failure in distributed systems, *Computers, IEEE Transactions on* **C-32**(9): 799–808.

Clermidy, F., Cassiau, N., Coste, N., Dutoit, D., Fantini, M., Ktenas, D., Lemaire, R. and Stefanizzi, L. [2011]. Reconfiguration of a 3gpp-lte telecommunication application on a 22-core noc-based system-on-chip, *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pp. 261–262.

Collet, J. H., Zajac, P., Psarakis, M. and Gizopoulos, D. [2011]. Chip self-organization and fault tolerance in massively defective multicore arrays, *IEEE Trans. Dependable Secur. Comput.* **8**(2): 207–217.

Constantinides, K., Mutlu, O., Austin, T. and Bertacco, V. [2007]. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation, *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, IEEE Computer Society, Washington, DC, USA, pp. 97–108.

Cornbaz, J., Fernandez, J., Lepley, T. and Sifakis, J. [2005]. Fine grain qos control for multimedia application software, *Proceedings of Design, Automation and Test in Europe, volume 2*, pp. 1038 – 1043.

Cornbaz, J., Fernandez, J., Sifakis, J. and Strus, L. [2007]. Using speed diagrams for symbolic quality management, *IEEE International Conference on Parallel and Distributed Processing Symposium - IPDPS*, pp. 1 – 8.

Cumming, P. [2003]. The TI OMAP Platform Approach to SoC, *in* G. Martin and H. Chang (eds), *Winning the SOC Revolution*, Kluwer Academic Publishers, pp. 97–118.

Dagum, L. and Menon, R. [1998]. Openmp: an industry standard api for shared-memory programming, *Computational Science Engineering, IEEE* **5**(1): 46–55.

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

Dally, W. J. and Towles, B. [2001]. Route packets, not wires: on-chip inteconnection networks, *Proceedings of the 38th annual Design Automation Conference*, DAC '01, ACM, New York, NY, USA, pp. 684–689.

David, P.-C. and Ledoux, T. [2003]. Towards a framework for self-adaptive component-based applications, *In DAIS'03, volume 2893 of LNCS*, Springer-Verlag, pp. 1 – 14.

David, R., Bogdan, P., Marculescu, R. and Ogras, U. [2011]. Dynamic power management of voltage-frequency island partitioned networks-on-chip using intel's single-chip cloud computer, *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pp. 257–258.

de Kock, E. A. [2002]. Multiprocessor mapping of process networks: A jpeg decoding case study, *System Synthesis, International Symposium on* **0**: 68–73.

de Kruijf, M., Nomura, S. and Sankaralingam, K. [2010]. Relax: An architectural framework for software recovery of hardware faults, *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, ACM, New York, NY, USA, pp. 497–508.

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. [2002]. A fast and elitist multiobjective genetic algorithm: Nsga-ii, *Evolutionary Computation, IEEE Transactions on* **6**(2): 182 –197.

DeHon, A., Quinn, H. M. and Carter, N. P. [2010]. Vision for cross-layer optimization to address the dual challenges of energy and reliability, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 1017–1022.

DeOrio, A., Fick, D., Bertacco, V., Sylvester, D., Blaauw, D., Hu, J. and Chen, G. [2012]. A reliable routing architecture and algorithm for nocs, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **31**(5): 726–739.

Derin, O., Cannella, E., Tuveri, G., Meloni, P., Stefanov, T., Fiorin, L., Raffo, L. and Sami, M. [2013]. A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project, *Microprocessors and Microsystems* **37**(6–7): 515–529.

Derin, O. and Diken, E. [2010]. A task-aware middleware for fault-tolerance and adaptivity of kahn process networks on network-on-chip, *ReCoSoC 2010: Proceedings of the 5th International Workshop on Reconfigurable Communication-centric System-on-Chips*, Karlsruhe, Germany, pp. 73–78.

Derin, O., Diken, E. and Fiorin, L. [2011]. A middleware approach to achieving fault-tolerance of kahn process networks on networks-on-chips, *International Journal of Reconfigurable Computing* **2011**(Article ID 295385): 15 pages. Selected Papers from the International Workshop on Reconfigurable Communication-centric Systems on Chips (ReCoSoC' 2010).

Derin, O. and Ferrante, A. [2009]. Enabling Self-adaptivity in Component-based Streaming Applications, *SIGBED Review* **6**(3). Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'09).

Derin, O., Ferrante, A. and Taddeo, A. V. [2009]. Coordinated management of hardware and software self-adaptivity, *Journal of Systems Architecture* **55**(3): 170 – 179. Challenges in self-adaptive computing (Selected papers from the Aether-Morpheus 2007 workshop), Accepted Manuscript, Available online 29 July 2008.

Derin, O. and Fiorin, L. [2014]. Towards a reliability-aware design flow for kahn process networks on noc-based multiprocessors, *Proceedings of the 10th Workshop on Dependability and Fault Tolerance (ARCS/VERFE'14)*, Lübeck, Germany, pp. 1–8.

Derin, O., Kabakci, D. and Fiorin, L. [2011]. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors, *NOCS '11: Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, ACM, Pittsburgh, Pennsylvania, USA, pp. 129–136.

Derin, O., Ramankutty, P. K., Meloni, P. and Cannella, E. [2012]. Towards self-adaptive kpn applications on noc-based mpsocs, *Advances in Software Engineering* **2012**(Article ID 172674): 13 pages.

Derin, O., Ramankutty, P. K., Meloni, P. and Tuveri, G. [2013]. A low overhead self-adaptation technique for kpn applications on noc-based mpsocs, *Proceedings of the 3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS) - Special Session on Self-Adaptive Networked Embedded Systems (SANES)*, Barcelona, Spain, pp. 262–269.

Dieter, W. R. and Lumpp Jr, J. E. [1999]. A user-level checkpointing library for posix threads programs, *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, IEEE Computer Society, Washington, DC, USA, pp. 224–.

Dobson, S., Sterritt, R., Nixon, P. and Hinchey, M. [2010]. Fulfilling the vision of autonomic computing, *Computer* **43**(1): 35 –41.

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M. and Johnson, D. B. [2002]. A survey of rollback-recovery protocols in message-passing systems, *ACM Comput. Surv.* **34**(3): 375–408.

Erbas, C., Cerav-Erbas, S. and Pimentel, A. [2006]. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design, *IEEE Tran. on Evolutionary Computation* **10**(3): 358–374.

Fekr, A. R., Khademzadeh, A., Janidarmian, M. and Bokharaei, V. S. [2010]. Bandwidth/fault tolerance/contention aware application-specific noc using pso as a mapping generator, *Proc. of The World Congress on Engineering*, pp. 247–252.

Fiorin, L. and Sami, M. [2013]. Fault-tolerant network interfaces for networks-on-chip, *IEEE Transactions on Dependable and Secure Computing* **99**(PrePrints): 1.

Foster, I., Roy, A. and Sander, V. [2000]. A quality of service architecture that combines resource reservation and application adaptation, *Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on* pp. 181–188.

Foutris, N., Psarakis, M., Gizopoulos, D., Apostolakis, A., Vera, X. and Gonzalez, A. [2010]. MT-SBST: Self-test optimization in multithreaded multicore architectures, *Test Conference (ITC), 2010 IEEE International*, pp. 1–10.

Gaisler, J. and Catovic, E. [2006]. Multi-Core Processor Based on LEON3-FT IP Core (LEON3-FT-MP), *DASIA 2006 - Data Systems in Aerospace*, Vol. 630 of *ESA Special Publication*, p. 76.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B. and Steenkiste, P. [2004]. Rainbow: Architecture-based self-adaptation with reusable infrastructure, *Computer* **37**(10): 46–54.

Geihs, K., Barone, P. and Eliassen, F. [2009]. A comprehensive solution for application-level adaptation, *Software – Practice & Experience* **39**(4): 385 – 422.

Gerstlauer, A., Haubelt, C., Pimentel, A. D., Stefanov, T. P., Gajski, D. D. and Teich, J. [2009]. Electronic system-level synthesis methodologies, *Trans. Comp.-Aided Des. Integ. Cir. Sys.* **28**(10): 1517–1530.

Gizopoulos, D. [April-June 2009]. Online periodic self-test scheduling for real-time processor-based systems dependability enhancement, *Dependable and Secure Computing, IEEE Transactions on* **6**(2): 152–158.

Gizopoulos, D., Psarakis, M., Adve, S., Ramachandran, P., Hari, S., Sorin, D., Meixner, A., Biswas, A. and Vera, X. [2011]. Architectures for online error detection and recovery in multicore processors, *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6.

Gizopoulos, D., Psarakis, M., Hatzimihail, M., Maniatakos, M., Paschalis, A., Raghunathan, A. and Ravi, S. [2008]. Systematic Software-Based Self-Test for Pipelined Processors, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* **16**(11): 1441–1453.

Gjørven, E., Eliassen, F., Lund, K., Eide, V. S. W. and Staehli, R. [2006]. Self-adaptive systems: A middleware managed approach, *in* A. Keller and J.-P. Martin-Flatin (eds), *SelfMan*, Vol. 3996 of *Lecture Notes in Computer Science*, Springer, pp. 15–27.

Grant, P., Saw, Y.-S. and Hannah, J. M. [1997]. Fuzzy rule-based MPEG video rate prediction and control, *Proceedings of the Eurasip ECASP Conference*, pp. 211–214.

Greene, J. W. and El Gamal, A. [1984]. Configuration of vlsi arrays in the presence of defects, *J. ACM* **31**(4): 694–717.

Gupta, S., Feng, S., Ansari, A., Blome, J. and Mahlke, S. [2008]. The stagenet fabric for constructing resilient multicore systems, *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, IEEE Computer Society, Washington, DC, USA, pp. 141–151.

Hafid, A. and v. Bochmann, G. [1998]. Quality-of-service adaptation in distributed multimedia applications, *Multimedia Systems* **6**(5): 299–315.

Haid et al., W. [2009]. Efficient execution of kahn process networks on multiprocessor systems using protothreads and windowed FIFOs, *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, IEEE, Grenoble, France, pp. 35–44.

Haid, W., Huang, K., Bacivarov, I. and Thiele, L. [2009]. Multiprocessor SoC software design flows, *IEEE Signal Processing Magazine* **26**: 64–71.

Hargrove, P. H. and Duell, J. C. [2006]. Berkeley lab checkpoint/restart (blcr) for linux clusters, *Technical Report LBNL-60520*, Lawrence Berkeley National Laboratory.

Hawthorne, M. J. and Perry, D. E. [2004]. Exploiting architectural prescriptions for self-managing, self-adaptive systems: a position paper, *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM, New York, NY, USA, pp. 75–79.

Howard, J., Dighe, S., Vangal, S., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., Droege, G., Lund-Larsen, T., Steibl, S., Borkar, S., De, V. and Van Der Wijngaart, R. [2011]. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling, *Solid-State Circuits, IEEE Journal of* **46**(1): 173 –183.

Howes, L. and Munshi, A. (eds) [2014]. *The OpenCL Specification Version: 2.0 rev. 26*, Khronos OpenCL Working Group.

Hu, J. and Marculescu, R. [2003]. Energy-aware mapping for tile-based noc architectures under performance constraints, *Proc. of the Asia and South Pacific Design Automation Conf.*, pp. 233 – 239.

Hu, J. and Marculescu, R. [2005]. Energy and performance aware mapping for regular noc architectures, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **24**(4): 551–562.

Huang et al., J. [2011]. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems, *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, ACM, New York, NY, USA, pp. 247–256.

Huang, L. and Xu, Q. [2010]. Energy-efficient task allocation and scheduling for multi-mode mpsocs under lifetime reliability constraint, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 1584–1589.

Huang, L., Ye, R. and Xu, Q. [2011]. Customer-aware task allocation and scheduling for multi-mode mpsocs, *Proceedings of the 48th Design Automation Conference*, DAC '11, ACM, New York, NY, USA, pp. 387–392.

Huang, L., Yuan, F. and Xu, Q. [2009]. Lifetime reliability-aware task allocation and scheduling for mpsoc platforms, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 51–56.

Huang, L., Yuan, F. and Xu, Q. [2011]. On task allocation and scheduling for lifetime extension of platform-based MPSoC designs, *Parallel and Distributed Systems, IEEE Transactions on* **22**(12): 2088 –2099.

Huebscher, M. C. and McCann, J. A. [2008]. A survey of autonomic computing—degrees, models, and applications, *ACM Comput. Surv.* **40**(3): 7:1–7:28.

Ibarra, O. H. and Kim, C. E. [1977]. Heuristic algorithms for scheduling independent tasks on nonidentical processors, *J. ACM* **24**: 280–289.

IBM ILOG CPLEX Optimizer [n.d.]. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

IEEE Standards Association [2012]. *IEEE Std 1666-2011, IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Computer Society.

ITRS [2009]. International technology roadmap for semiconductors - design chapter.
URL: *http://www.itrs.net*

Izosimov, V., Pop, P., Eles, P. and Peng, Z. [2012]. Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs, *ACM Trans. Embed. Comput. Syst.* **11**(3): 61:1–61:35.

Jaber, M., Combaz, J. and Strus, L. [2008]. Using neural networks for quality management, *IEEE International Conference on Emerging Technologies and Factory Automation - ETFA*, pp. 1441 – 1448.

Jantsch, A. [2003]. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Jena, R. K. and Mahanti, P. K. [2008]. Design space exploration of network-on-chip - a system level approach, *Int. J. of Computing and ICT Research* **2**: 17–25.

Józwiak, L. [2006]. Life-inspired systems and their quality-driven design., *in* W. Grass, B. Sick and K. Waldschmidt (eds), *ARCS*, Vol. 3894 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.

Kahn, G. [1974]. The semantics of a simple language for parallel programming, *in* J. L. Rosenfeld (ed.), *Information Processing '74: Proceedings of the IFIP Congress*, North-Holland, New York, NY, pp. 471–475.

Kang, S.-H., Yang, H., Kim, S., Bacivarov, I., Ha, S. and Thiele, L. [2014a]. Reliability-aware mapping optimization of multi-core systems with mixed-criticality, *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 327:1–327:4.

Kang, S.-h., Yang, H., Kim, S., Bacivarov, I., Ha, S. and Thiele, L. [2014b]. Static mapping of mixed-critical applications for fault-tolerant mpsocs, *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, ACM, New York, NY, USA, pp. 31:1–31:6.

Karsai, G., Lédeczi, Á., Sztipanovits, J., Péceli, G., Simon, G. and Kovácsházy, T. [2001]. An approach to self-adaptive software based on supervisory control, *IWSAS*, pp. 24–38.

Khalili, F. and Zarandi, H. [2012]. A fault-aware low-energy spare core allocation in networks-on-chip, *NORCHIP, 2012*, pp. 1–4.

Kogel, T., Leupers, R. and Meyr, H. [2006]. *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms*, Springer Netherlands.

Koren, I. and Krishna, C. M. [2007]. *Fault Tolerant Systems*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kramer, J. and Magee, J. [2007]. Self-managed systems: an architectural challenge, *FOSE '07: 2007 Future of Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 259–268.

Krishna, C. M. [2014]. Fault-tolerant scheduling in homogeneous real-time systems, *ACM Comput. Surv.* **46**(4): 48:1–48:34.

Kwon, S., Kim, Y., Jeun, W.-C., Ha, S. and Paek, Y. [2008]. A retargetable parallel-programming framework for mpsoc, *ACM Trans. Des. Autom. Electron. Syst.* **13**: 39:1–39:18.

Lan, Z. and Li, Y. [2008]. Adaptive fault management of parallel applications for high-performance computing, *IEEE TRANSACTIONS ON COMPUTERS* **57**(12).

Le Beux, S., Bois, G., Nicolescu, G., Bouchebaba, Y., Langevin, M. and Paulin, P. [2010]. Combining mapping and partitioning exploration for noc-based embedded systems, *J. Syst. Archit.* **56**: 223–232.

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

Lee, E. A. and Messerschmitt, D. G. [1987a]. Synchronous data flow, *Proceedings of the IEEE* **75**(9): 1235–1245.

Lee, E. and Messerschmitt, D. [1987b]. Synchronous data flow, *Proceedings of the IEEE* **75**(9): 1235–1245.

Lee, E. and Sangiovanni-Vincentelli, A. [1998]. A framework for comparing models of computation, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **17**(12): 1217–1229.

Lee, K., Shrivastava, A., Kim, M., Dutt, N. and Venkatasubramanian, N. [2008]. Mitigating the impact of hardware defects on multimedia applications: A cross-layer approach, *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, ACM, New York, NY, USA, pp. 319–328.

Lei, T. and Kumar, S. [2003]. A two-step genetic algorithm for mapping task graphs to a network on chip architecture, *Proc. of Euromicro Symposium on Digital System Design*, pp. 180 – 187.

Lemos, R., Giese, H., Müller, H. A. and Shaw, M. [2013]. *Software Engineering for Self-adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, Springer Berlin Heidelberg.

Li, B. and Nahrstedt, K. [1998]. An Open Task Control Model for Quality of Service Adaptation, *Proceedings of the 14th International Conference of Advanced Science and Technology (ICAST 98)*, Naperville, Illinois, pp. 29–41.

Li, B. and Nahrstedt, K. [1999]. A Control-Based Middleware Framework for Quality-of-Service Adaptations, *IEEE Journal on Selected Areas in Communications* **17**(9): 1632–1650.

Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S. and Zhou, Y. [2008]. Understanding the propagation of hard errors to software and implications for resilient system design, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, ACM, New York, NY, USA, pp. 265–276.

Li, X. and Yeung, D. [2007]. Application-level correctness and its impact on fault tolerance, *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, IEEE Computer Society, Washington, DC, USA, pp. 181–192.

Lieverse, P., Stefanov, T., van der Wolf, P. and Deprettere, E. [2001]. System level design with spade: an m-jpeg case study, *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, pp. 31 –38.

Litzkow, M., Tannenbaum, T., Basney, J. and Livny, M. [1997]. Checkpoint and migration of UNIX processes in the Condor distributed processing system, *Technical Report UW-CS-TR-1346*, University of Wisconsin - Madison Computer Sciences Department.

Marcon, C., Calazans, N., Moraes, F., Susin, A., Reis, I. and Hessel, F. [2005]. Exploring noc mapping strategies: an energy and timing aware technique, *Proc. of DATE*, pp. 502–507.

Marwedel, P. [2011]. *Embedded System Design*, Springer Netherlands.

Meixner, A., Bauer, M. E. and Sorin, D. [2007]. Argus: Low-cost, comprehensive error detection in simple cores, *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, IEEE Computer Society, Washington, DC, USA, pp. 210–222.

Meixner, A. and Sorin, D. [2008]. Detouring: Translating software to circumvent hard faults in simple cores, *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 80–89.

Meloni, P., Secchi, S. and Raffo, L. [2010]. An fpga-based framework for technology-aware prototyping of multicore embedded architectures, *IEEE Embedded Systems Letters* **2**(1): 5–9.

Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F. and Dutoit, D. [2012]. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications, *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, ACM, New York, NY, USA, pp. 1137–1142.

Milojičić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S. [2000]. Process migration, *ACM Comput. Surv.* **32**: 241–299.

Mitra, S., Brelsford, K. and Sanda, P. [2010]. Cross-layer resilience challenges: Metrics and optimization, *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 1029–1034.

Modarressi, M. and Sarbazi-Azad, H. [2007]. Power-aware mapping for reconfigurable noc architectures, *25th Int. Conf. on Computer Design*, pp. 417 –422.

Mossé, D., Melhem, R. and Ghosh, S. [2003]. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation, *IEEE Trans. Softw. Eng.* **29**(8): 752–767.

Mukherjee, S. [2008]. *Architecture Design for Soft Errors*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Murali, S. and De Micheli, G. [2004]. Bandwidth-constrained mapping of cores onto noc architectures, *Proc. of the Design Automation and Test Europe Conf.*, Vol. 2, pp. 896–901.

Nadezhkin, D., Meijer, S., Stefanov, T. and Deprettere, E. [2009]. Realizing FIFO Communication when Mapping Kahn Process Networks onto the Cell, *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, Springer-Verlag, Berlin, Heidelberg, pp. 308–317.

Nakano, J., Montesinos, P., Gharachorloo, K. and Torrellas, J. [2006]. Revivei/o: efficient handling of i/o in highly-available rollback-recovery servers, *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 200–211.

Neema, S. and Lédeczi, Á. [2001]. Constraint-guided self-adaptation, *in* R. Laddaga, P. Robertson and H. E. Shrobe (eds), *IWSAS*, Vol. 2614 of *Lecture Notes in Computer Science*, Springer, pp. 39–51.

Nejad, A. B., Goossens, K., Walters, J. and Kienhuis, B. [2009]. Mapping kpn models of streaming applications on a network-on-chip platform, *ProRISC 2009: Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits*.

Neuendorffer, S. and Lee, E. [2004]. Hierarchical reconfiguration of dataflow models, *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 179–188.

Nichols, B., Buttlar, D. and Farrell, J. P. [1996]. *Pthreads Programming*, O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Nieuwland, A., Kang, J., Gangwal, O. P., Sethuraman, R., Busá, N., Goossens, K., Peset Llopis, R. and Lippens, P. [2002]. C-heap: A heterogeneous multiprocessor architecture template and scalable and flexible protocol for the design of embedded signal processing systems, *Design Automation for Embedded Systems* **7**: 233–270. 10.1023/A:1019782306621.

Nikolopoulos, D. S., Ayguadé, E., Papatheodorou, T. S., Polychronopoulos, C. D. and Labarta, J. [2001]. The trade-off between implicit and explicit data distribution in shared-memory programming paradigms, *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, ACM, New York, NY, USA, pp. 23–37.

Nikolov et al., H. [2008]. Daedalus: Toward composable multimedia MP-SoC design, *45th ACM/IEEE Design Automation Conference, 2008 (DAC 2008).*, pp. 574–579.

Nikolov, H., Rao, A., Deprettere, E. F., Nandy, S. K. and Narayan, R. [2009]. A h.264 decoder: a design style comparison case study, *Proceedings of the 43rd Asilomar conference on Signals, systems and computers*, Asilomar'09, IEEE Press, Piscataway, NJ, USA, pp. 236–242.

Nikolov, H., Stefanov, T. and Deprettere, E. [2008]. Systematic and Automated Multiprocessor System Design, Programming, and Implementation, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **27**(3): 542–555.

Nollet, V., Verkest, D. and Corporaal, H. [2010]. A safari through the mpsoc run-time management jungle, *Signal Processing Systems* **60**(2): 251–268.

Nvidia [2014]. *Compute unified device architecture (CUDA) C programming guide*, NVidia.

Oliveira, J. A. D. and van Antwerpen, H. [2003]. The Philips Nexperia digital video platform, *in* G. Martin and H. Chang (eds), *Winning the SOC Revolution*, Kluwer Academic Publishers, pp. 67–96.

Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. and Wolf, A. [1999]. An architecture-based approach to self-adaptive software.

Pacheco, P. S. [1996]. *Parallel Programming with MPI*, Morgan Kaufmann.

Patnaik, L. M. and Iyer, K. V. [1986]. Load-leveling in fault-tolerant distributed computing systems, *IEEE Transactions on Software Engineering* **12**(4): 554–560.

Pellegrini, A., Smolinski, R., Chen, L., Fu, X., Hari, S. K. S., Jiang, J., Adve, S. V., Austin, T. and Bertacco, V. [2012]. Crashtest'ing swat: Accurate, gate-level evaluation of symptom-based resiliency solutions, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, EDA Consortium, San Jose, CA, USA, pp. 1106–1109.

Pinello, C., Carloni, L. and Sangiovanni-Vincentelli, A. [2008]. Fault-tolerant distributed deployment of embedded control software, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **27**(5): 906–919.

Piscitelli, R. and Pimentel, A. [2012]. Design space pruning through hybrid analysis in system-level design space exploration, *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 781–786.

Pittau, M., Alimonda, A., Carta, S. and Acquaviva, A. [2007]. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation, *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pp. 59–64.

Powell, M. D., Biswas, A., Gupta, S. and Mukherjee, S. S. [2009]. Architectural core salvaging in a multi-core processor for hard-error tolerance, *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, ACM, New York, NY, USA, pp. 93–104.

Pradhan, D. and Vaidya, N. [1994]. Roll-forward checkpointing scheme: a novel fault-tolerant architecture, *Computers, IEEE Transactions on* **43**(10): 1163–1174.

Prvulovic, M., Zhang, Z. and Torrellas, J. [2002]. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, IEEE Computer Society, Washington, DC, USA, pp. 111–122.

Reick, K., Sanda, P. N., Swaney, S., Kellington, J. W., Mack, M., Floyd, M. and Henderson, D. [2008]. Fault-tolerant design of the ibm power6 microprocessor, *IEEE Micro* **28**(2): 30–38.

Renesas [2013]. Semiconductor reliability handbook.
URL: *http://www.renesas.com*

Rezaei, M., Akhbardeh, A., Hannuksela, M. and Gabbouj, M. [2006]. Fuzzy rate controller for variable bitrate video in mobile applications, *Communications, 2006. ICC '06. IEEE International Conference on*, Vol. 7, pp. 3197–3201.

Romanescu, B. F. and Sorin, D. J. [2008]. Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults, *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, ACM, New York, NY, USA, pp. 43–51.

Salehie, M. and Tahvildari, L. [2009]. Self-adaptive software: Landscape and research challenges, *ACM Trans. Auton. Adapt. Syst.* **4**(2): 14:1–14:42.

Salfner, F., Lenk, M. and Malek, M. [2010]. A survey of online failure prediction methods, *ACM Comput. Surv.* **42**(3): 10:1–10:42.

Saraswat, P. K., Pop, P. and Madsen, J. [2010]. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems, *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, IEEE Computer Society, Washington, DC, USA, pp. 89–98.

Sastry Hari, S. K., Li, M.-L., Ramachandran, P., Choi, B. and Adve, S. V. [2009]. mswat: Low-cost hardware fault detection and diagnosis for multicore systems, *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, ACM, New York, NY, USA, pp. 122–132.

Schantz, R. E., Loyall, J. P., Rodrigues, C. and Schmidt, D. C. [2006]. Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems, *Software–Practice & Experience* **36**(11-12): 1189–1208.

Schmeck, H. [2005]. Organic computing - a new vision for distributed embedded systems, *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pp. 201–203.

Schmoll, F., Heinig, A., Marwedel, P. and Engel, M. [2013]. Improving the fault resilience of an h.264 decoder using static analysis methods, *ACM Trans. Embed. Comput. Syst.* **13**(1s): 31:1–31:27.

Scholzel, M., Koal, T. and Vierhaus, H. [2012]. An adaptive self-test routine for in-field diagnosis of permanent faults in simple risc cores, *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pp. 312–317.

Schuchman, E. and Vijaykumar, T. N. [2005]. Rescue: A microarchitecture for testability and defect tolerance, *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, IEEE Computer Society, Washington, DC, USA, pp. 160–171.

Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K. and Stodghill, P. [2004]. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs, *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, IEEE Computer Society, Washington, DC, USA, pp. 38–.

Shivakumar, P., Keckler, S., Moore, C. and Burger, D. [2003]. Exploiting microarchitectural redundancy for defect tolerance, *Computer Design, 2003. Proceedings. 21st International Conference on*, pp. 481–488.

Singh, A. K., Shafique, M., Kumar, A. and Henkel, J. [2013]. Mapping on multi/many-core systems: Survey of current and emerging trends, *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, ACM, New York, NY, USA, pp. 1:1–1:10.

Singh, S., Han, J.-Y. and Stefanek, G. [1991]. A heuristic approach to load sharing in fault-tolerant distributed systems, *Circuits and Systems, 1991., Proceedings of the 34th Midwest Symposium on*, pp. 629–632 vol.2.

Sinnamon, R. M. [1996]. *Binary Decision Diagrams for Fault Tree Analysis*, PhD thesis, Loughborough University.

Smith, J. M. [1988]. A survey of process migration mechanisms, *SIGOPS Oper. Syst. Rev.* **22**: 28–40.

Sorin, D. J. [2009]. *Fault Tolerant Computer Architecture*, Morgan and Claypool Publishers.

Sorin, D. J., Martin, M. M. K., Hill, M. D. and Wood, D. A. [2002]. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, IEEE Computer Society, Washington, DC, USA, pp. 123–134.

Srinivasan, J. [2006]. *Lifetime reliability aware microprocessors*, PhD thesis, University of Illinois at Urbana-Champaign.

Srinivasan, J., Adve, S. V., Bose, P. and Rivers, J. A. [2005]. Exploiting structural duplication for lifetime reliability enhancement, *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, IEEE Computer Society, Washington, DC, USA, pp. 520–531.

Srinivasan, K. and Chatha, K. [2005]. A technique for low energy mapping and routing in network-on-chip architectures, *Proc. of the Int. Symposium on Low Power Electronics and Design*, pp. 387 – 392.

Srinivasan, K., Chatha, K. S. and Konjevod, G. [2006]. Linear-programming-based techniques for synthesis of network-on-chip architectures, *IEEE Trans. Very Large Scale Integr. Syst.* **14**: 407–420.

Stefanov, T., Kienhuis, B. and Deprettere, E. [2002]. Algorithmic transformation techniques for efficient exploration of alternative application instances, *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, ACM, New York, NY, USA, pp. 7–12.

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B. and Deprettere, E. [2004]. System design using kahn process networks: The compaan/laura approach, *Design, Automation and Test in Europe Conference and Exhibition* **1**: 10340.

Stralen, P. v. and Pimentel, A. [2012]. A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs, *Proceedings of CODES+ISSS*, pp. 393–402.

The Multicore Association [2011]. Multicore communication API (MCAPI) specification v2.015.
URL: *http://www.multicore-association.org*

Thiele, L., Bacivarov, I., Haid, W. and Huang, K. [2007]. Mapping applications to tiled multiprocessor embedded systems, *Seventh Int. Conf. on Application of Concurrency to System Design*, pp. 29 –40.

Thies, W. and Amarasinghe, S. [2010]. An empirical characterization of stream programs and its implications for language and compiler design, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 365–376.

van der Wolf, P., Lieverse, P., Goel, M., La Hei, D. and Vissers, K. [1999]. An mpeg-2 decoder case study as a driver for a system level design methodology, *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, ACM, New York, NY, USA, pp. 33–37.

Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N. and Borkar, S. [2008]. An 80-tile sub-100-w teraflops processor in 65-nm cmos, *Solid-State Circuits, IEEE Journal of* **43**(1): 29 –41.

Vaughan, F. and Munro, D. [2000]. Self-adaptive compliant persisent architectures, *Proceedings of the Seventh Integrated Data Environments - Australia (IDEA'07) Workshop*, pp. 5–10.

Verdoolaege, S. [2010]. *Handbook on signal processing systems*, Springer, chapter Polyhedral process networks.

Verdoolaege, S., Nikolov, H. and Stefanov, T. [2007]. pn: A Tool for Improved Derivation of Process Networks, *EURASIP J. Embedded Syst.* **2007**: 19–19.

Vesely, W. E. [1971]. Reliability and fault tree applications at the NRTS, *Nuclear Science, IEEE Transactions on* **18**(1): 472 –480.

Vesely, W. E., Goldberg, F. F., Roberts, N. H. and Haasl, D. F. [1981]. Fault tree handbook, *Technical report*, DTIC Document.

Vrba, Z., Halvorsen, P. and Griwodz, C. [2009]. Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors, *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, pp. 639 –644.

Walter, I., Cidon, I., Kolodny, A. and Sigalov, D. [2009]. The era of many-modules soc: revisiting the noc mapping problem, *2nd Int. Workshop on Network on Chip Architectures*, pp. 43 –48.

Wang, X., Yang, M., Jiang, Y. and Liu, P. [2010]. A power-aware mapping approach to map ip cores onto nocs under bandwidth and latency constraints, *ACM Trans. Archit. Code Optim.* **7**: 1–30.

Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown, J. and Agarwal, A. [2007]. On-chip interconnection architecture of the tile processor, *Micro, IEEE* **27**(5): 15 –31.

Xilinx [2010]. Embedded processor block in virtex-5 fpgas, `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

Xu, J. and Randell, B. [1996]. Roll-forward error recovery in embedded real-time systems, *Proc. Int. Conf. on Parallel and Distributed Systems*, pp. 414–421.

Yetim, Y., Martonosi, M. and Malik, S. [2013]. Extracting useful computation from error-prone processors for streaming applications, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp. 202–207.

Yi, Y., Han, W., Zhao, X., Erdogan, A. T. and Arslan, T. [2009]. An ilp formulation for task mapping and scheduling on multi-core architectures, *Proc. of the Design Automation and Test Europe Conf.*, pp. 33–38.

Zadeh, L. [1965]. Fuzzy sets, *Information and Control* **8**: 338–353.

Zhai, J., Nikolov, H. and Stefanov, T. [2011]. Modeling adaptive streaming applications with parameterized polyhedral process networks, *Proceedings of the 48th Design Automation Conference*, ACM, pp. 116–121.

Zhang, J. and Cheng, B. H. C. [2006]. Model-based development of dynamically adaptive software, *ICSE '06: Proceeding of the 28th international conference on Software engineering*, ACM, New York, NY, USA, pp. 371–380.

Zhang, L., Han, Y., Xu, Q. and Li, X. [2008]. Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, ACM, New York, NY, USA, pp. 891–896.

Zhou, W., Zhang, Y. and Mao, Z. [2006]. Pareto based multi-objective mapping ip cores onto noc architectures, *IEEE Asia Pacific Conf. on Circuits and Systems*, pp. 331 –334.

Zrida, H., Abid, M., Ammri, A. and Jemai, A. [2008]. A yapi-kpn parallel model of a h264/avc video encoder, *Research in Microelectronics and Electronics, 2008. PRIME 2008. Ph.D.*, pp. 109 –112.