UNIVERSITY OF SOUTHAMPTON

Investigation into Low Power and Reliable System-on-Chip Design

by

Rishad Ahmed Shafik

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Medicine
School of Electronics and Computer Science

June 2010

UNIVERSITY OF SOUTHAMPTON
School of Electronics and Computer Science

Doctor of Philosophy

Investigation into Low Power and Reliable System-on-Chip Design

ABSTRACT

Rishad Ahmed Shafik

It is likely that the demand for multiprocessor system-on-chip (MPSoC) with low power consumption and high reliability in the presence of soft errors will continue to increase. However, low power and reliable MPSoC design is challenging due to conflicting trade-off between power minimisation and reliability objectives. This thesis is concerned with the development and validation of techniques to facilitate effective design of low power and reliable MPSoCs. Special emphasis is placed upon system-level design techniques for MPSoCs with voltage scaling enabled processors highlighting the trade-offs between performance, power consumption and reliability.

An important aspect in the system-level design is to validate reliability in the presence of soft errors through simulation technique. The first part of the thesis addresses the development of a SystemC fault injection simulator based on a novel fault injection technique. Using MPEG-2 decoder and other examples, it is shown that the simulator benefits from minimum design intrusion and high fault representation. The simulator is used throughout the thesis to facilitate the study of reliability of MPSoC.

On-chip communication architecture plays a vital role in determining the performance and reliability of MPSoCs. The second part of the thesis focuses on comparative study between two types of on-chip communication architectures: network-on-chip (NoC) and advanced microprocessor bus architecture (AMBA). The comparisons are carried out using real application traffic based on MPEG-2 video decoder demonstrating the trade-off between performance and reliability.

The third part of the thesis concentrates on low power and reliable system-level design techniques. Two new techniques are presented, which are capable of generating optimised designs in terms of low power consumption and reliability. The first technique demonstrates a power minimisation technique through appropriate voltage scaling of the MPSoC cores, such that real-time constraints are met and reliability is maintained at acceptable-level. The second technique deals with joint optimisation of power minimisation and reliability improvement for time-constrained MPSoCs. Extensive experiments are conducted for these two new techniques using different applications, including MPEG-2 video decoder. It is shown that the proposed techniques give significant power reduction and reliability improvement compared to existing techniques.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, *Rishad Ahmed Shafik*, declare that the thesis entitled

Investigation into Low Power and Reliable System-on-Chip Design

and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that:

1. This work was done during the period of candidature of the intended qualification at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this has been clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself or jointly with others, I have made clear exactly what was done by others and what have been contributed by myself;
7. Parts of this work have been published as listed in page 3, Section 1.2 (Chapter 1).

Signed:

Date:

List of Acronyms

ABB	Adaptive base bias
AHB	Advanced high-performance bus
AMBA	Advanced microprocessor bus architecture
APB	Advanced peripheral bus
ASA	Adaptive simulated annealing
ASB	Advanced system bus
ASIC	Application-specific integrated circuit
AXI	Advanced extensible interface
BILP	Binary integer linear programming
CIF	Common intermediate format
CLICHE	Chip-level integration of communicating heterogeneous elements
DCT	Discrete cosine transformation
DRAM	Dynamic random access memory
DSP	Digital signal processor
DTU	Data transaction unit
DUT	Device under test
DVS	Dynamic voltage scaling
EDA	Electronic design automation
ESL	Electronic system-level
FER	Frame error ratio/rate
FI	Fault injection
FIFO	First-in first-out
FIR	Fault injection rate
FIT	Fault injection time

FLIT	Flow control unit
FPGA	Field programmable gate array
HDL	Hardware description language
HW	Hardware
HW/SW	Hardware / software
IDCT	Inverse discrete cosine transformation
ILP	Integer linear programming
ISQ	Inverse scan and quantisation
MC	Motion Compensation
MCU	Microcontroller unit
MPARM	Multiprocessor ARM
MPEG	Moving picture experts group
MPSoC	Multiprocessor system-on-chip
MTBF	Mean-time-between-failure
MTTF	Mean-time-to-failure
NI	Network interface
NoC	Network-on-chip
NTSC	National television system committee
OSCI	Open SystemC initiative
P2P	Point-to-Point
PAL	Phase alternating line
PE	Processing element
PSNR	Peak signal-to-noise ratio
QCIF	Quarter common intermediate format
RAM	Random access memory
RHEL	Red Hat Enterprise Linux
RTEMS	Real-time executive for embedded systems
RTOS	Real-time operating system
RTL	Register transfer level
SA	Simulated Annealing
SER	Soft error rate

SEU	Single-event upset
SoC	System-on-chip
SRAM	Static random access memory
SW	Software
VLD	Variable length decoder
VLSI	Very large scale integration

List of Symbols

$d^{j,k}$	Communication cost between j-th and k-th task, in clock cycles
C	Number of processing cores in MPSoC architecture
D	Degree of concurrency of cores
D_r	MPEG decoding rate, frames per second
D_{rated}	Specified MPEG frame rate, frames per second
f	Operating clock frequency, in MHz
f_{eff}	Effective multiprocessor frequency, in MHz
G	Task graph with a set of computational and communication tasks
M	Number of MPEG frames in a video sequence
N	Number of tasks in a task graph
K	Number of switches between communicating NoC cores
L	Number of pixels in an MPEG frame
L_{ch}	Channel latency, in clock cycles
P	Power consumption, in mW
p_B	Fault probability per bit
P_{dyn}	Dynamic power consumption, in mW
P_{leak}	Leakage power consumption, in mW
r_j	Register usage of j-th task in a task graph, in bits per cycle
R_i	Register usage of i-th processing core, in bits per cycle
R	Overall register usage of an MPSoC, in bits per cycle
S	Number of scaling levels in a voltage scaling-enabled system
t_j	Computational cost of j-th task, in clock cycles
T_A	Total application time, in clock cycles
T_E	Processor execution time, in clock cycles

T_M	Multiprocessor execution time, in clock cycles
T_{NP}	Non-processing time in a core, in clock cycles
T_P	Processing time in a core, in clock cycles
T_{MB}	Per macroblock execution time
T_i^{I-I}	Idle-idle transition times of i-th processing core of an MPSoC, in clock cycles
T_i	Execution time of i-th processing core of an MPSoC, in clock cycles
V_{dd}	Operating supply voltage, in volts
Z	Optimisation score function
Z_P	Optimisation score function for power
Z_Γ	Optimisation score function for SEUs experienced
α_i	Activity factor of i-th processing core within an MPSoC
$\mathcal{E}$	Set of communication edges in a task graph
$\mathcal{F}$	Total number of SEUs experienced by computation cores and communication interconnects
$\mathcal{F}_{comp}$	Number of SEUs experienced during computation
$\mathcal{F}_{comm}$	Number of SEUs experienced during communication
Γ	Number of SEUs experienced
λ	Soft error rate after scaling, in SEUs per bit per cycle
λ	Base soft error rate before scaling, in SEUs per bit per cycle
$\mathcal{N}$	Number of data transaction units for inter-core communication within an application
$\mathcal{M}$	Number of inter-core communication links within the MPSoC
Ω	Application-level correctness
$\mathcal{V}$	Set of computational nodes in a task graph
ρ	Ratio of register usage (R_i) of a processing core to the overall register usage (R) of an MPSoC
σ	Core efficiency of a processing core within an MPSoC
τ	Duration of a bit upset for transient faults, in cycles

Acknowledgements

I would like to express my sincere gratitude to my supervisor, *Professor Bashir M. Al-Hashimi*, for the privilege of working under his supervision. I am deeply indebted to him for his constant support and guidance throughout this research. His comments and suggestions at different phases of work and related writings has been extra-ordinary. The intellectual freedom, encouragements and the time-to-time nudging in the right direction from his side have made my time very useful and productive. I also gratefully acknowledge and appreciate the valuable advice and encouragement I received from *Professor Eric Rogers*. His comments and feedback at different stages of PhD have been truly enlightening. *Dr. Jeff Reeve* deserves profound appreciation for his constructive comments during nine-month, MPhil/PhD transfer and PhD examinations.

Given this opportunity, I would like to thank *Professor Sandip Kundu* (University of Massachusetts, Amherst, USA) and *Professor Krishnendu Chakrabarty* (Duke University, USA) for their very useful feedback regarding this work. Their insightful comments and suggestions helped me address the different challenges more efficiently. I have also received useful suggestions about my work from colleagues in the research group, *Dr. Paul Rosinger*, *Dr. Alireza Ejlali* and *Dr. Saqib Khursheed*. *Lavina Jain* merits a worthy mention for her untiring efforts to develop the NIRGAM simulator, which has been used in this research. I am truly grateful to all of them. I am also thankful to my fellow mates in the lab, *Amit*, *Mustafa*, *Simon*, *Sheng*, *Shida* and *Jatin* to name a few, for the help, knowledge and encouragement I received from them over the years. Part of my work has been funded by Engineering and Physical Sciences Research Council (EPSRC) and I am indebted to them.

To my parents, I owe my deepest appreciation and admiration. Throughout my life, they have encouraged me in everything I ever wanted to do. Even at great hardship to themselves, they made sure I had every opportunity they could possibly give me. My brothers, *Sadrul* and *Rafique*, have also been outstanding with their support from time-to-time. I hope that my accomplishments will make all of them proud. Last, but not the least, I would like to thank my wife, *Sonia*, whole-heartedly for her enormous patience, emotional support and inspiration.

to my parents.....

Chapter 1

Introduction

Over the last decade, the popularity of portable electronic devices has increased significantly. These devices are realised by interconnecting various system components into a single integrated circuit, called system-on-chip (SoC). The methodology of component interconnection in an SoC is defined by an on-chip communication architecture. Development of efficient on-chip communication architecture is a crucial design issue as it greatly influences the underlying performance of an SoC. Since most of these devices are battery-powered, low power consumption is a prime design objective to extend battery life. Recently, reliability is emerging as another design requirement to operate in the presence of soft errors (i.e. transient faults caused by radiation and harsh operating environment). However, low power consumption and reliability are conflicting design objectives as power minimisation techniques cause substantial increase in the number of soft errors. Hence, design of low power and reliable SoCs is a complex and challenging task. To produce optimised designs, appropriate design flows are needed. This thesis addresses the above challenges through investigation into design and development of efficient, low power and reliable SoCs. The remainder of this chapter is organised as follows. Section 1.1 underlines the motivations of the research carried out in this thesis and Section 1.2 presents an overview of the thesis and its contributions.

1.1 Motivations and Scope of the Thesis

Embedded systems are making their way into more and more devices, from hand-held gadgets to household appliances, and from mobile devices to cars. The current trend is that this growth will continue and the market is expected to experience a three-fold rise in the demand from 2009 to 2014 [1]. This growing demand has led the designers to reduce cost and improve system performance through integration of more functionality into a single chip. The continued increase in device integration has become possible due

to CMOS transistor scaling and miniaturisation described by the Moore's Law [2, 3].

Traditionally, embedded systems are designed with single-processor chips. To accommodate more functionality with increased complexity of applications, multiprocessor system-on-chip (MPSoC) is emerging as a popular embedded systems platform. MPSoC consists of multiple processing elements on a single piece of silicon, each with specific functionality reflecting the need of the expected application domain. The inclusion of multiple processing elements in MPSoCs has a number of benefits, including parallel processing, low clock speed and low power consumption [4]. However, the design of the current and future MPSoCs presents a number of challenges [5]. A core issue in the design of MPSoCs is on-chip communication architecture, which affects the performance of the system [6]. Also, since a vast majority of today's embedded systems are battery powered, a prime design objective is to minimise power consumption to extend battery life of these systems. However, power minimisation is reported to cause exponential increase in the number of soft errors, particularly that of single-event upsets (SEUs) caused by cosmic or electromagnetic radiation [7, 8, 9]. The existence of these errors highlighting the impact of technology scaling and operating environments has been investigated in a number of academic [10] and industrial studies [11, 12]. Therefore, reliability of MPSoCs in the presence of soft errors is an emerging design challenge in addition to high performance and low power consumption requirements. A number of design approaches have been proposed by researchers over the years addressing various issues related to low power and reliable design of MPSoCs, such as [13, 14, 15, 16].[1]

This research addresses the following challenges in the design of efficient, low power and reliable MPSoCs:

1. To date there has been good progress in developing efficient on-chip communication architectures. Advanced microprocessor bus architecture (AMBA) is today's dominant, industrial standard on-chip communication architecture [17]. AMBA enables multiple processing elements to be incorporated in an MPSoC through shared communication bus architecture [18]. Network-on-Chip (NoC) is an emerging paradigm for on-chip communication architecture in MPSoCs. It enables multiprocessing through integration and spatial multiplexing of communication interconnects among processing elements [19]. Over the years a number of studies have been carried out showing comparisons between AMBA and NoC in terms of performance, power and area, such as [20, 21, 22]. However, most of these studies are based on synthetic applications (random task graphs). Also, with reliability as an emerging design challenge, no study has been reported so far showing the impact of AMBA and NoC based on-chip communication architectures on MPSoC reliability. To understand the benefits and shortcomings of AMBA and NoC in terms of performance and reliability and to facilitate development of efficient

[1] A review of related previous works carried out is presented in Section 2.5, Chapter 2).

on-chip communication architecture, further investigations are needed using real application traffic.

2. There is growing interest in evaluating the impact of soft errors on MPSoCs at application-level rather than architectural-level. Such evaluation at application-level, often known as application-level correctness, has been reported to enable low-cost fault-tolerant design techniques, particularly in multimedia applications [23, 24]. To achieve low-cost power reduction the impact of system-level power minimisation on application-level correctness needs to be investigated. Currently no such study exists that address the relationship between system-level power minimisation and application-level correctness. Such relationship can be effectively employed to reduce power consumption, whilst maintaining acceptable application-level correctness and meeting the real-time performance constraints.

3. Traditionally, low power and reliable MPSoC design is carried out through power-aware fault tolerance techniques considering low power and reliability as two separate objectives. To find effective design optimisation technique with low power and improved reliability as a joint objective, further studies are needed to understand the reliability of applications, particularly from system- and application-level design perspective. Application task mapping is one such design step of applications, which is concerned with distribution of computational and communication tasks among available resources in an MPSoC. Currently there is no study into the impact of the application task mapping on the reliability of application in the presence of soft errors. Such investigation is expected to facilitate soft error-aware design optimisation technique for MPSoCs such that both power minimisation and reliability improvement can be achieved at the same time.

4. To study the above challenges using simulation technique, appropriate tools for profiling power, performance and reliability need to be employed. A number of academic and industrial simulation tools have been developed so far that can effectively profile performance and power, such as MPARM [21], NIRGAM [25] and PrimeTime [26]. However, currently there is a lack of effective fault injection simulators based on system-level specification language, such as SystemC, offering high fault representation and minimum design intrusion.

1.2 Thesis Overview and Contributions

This thesis presents novel and effective techniques for the design of low power and reliable MPSoCs. An overview of the following chapters highlighting their contributions follows. In Chapter 2, the fundamental concepts used in this work are introduced and a brief review of related previous works is presented (further reviews of related works

are presented in each of the following chapters). Chapter 3 presents a prototype fault injection simulator using a novel fault injection technique. The fault injection is carried out through simulation command approach using SystemC description of the device under test. Comparisons with recently reported SystemC fault injection techniques show that the proposed fault simulator benefits from high fault representation, fast simulation speed and flexibility with different fault types and probabilities. The simulator is validated using an MPEG-2 video decoder and other examples. Chapter 4 highlights comparative performance and reliability analysis between traditional advanced microprocessor bus architecture (AMBA) and emerging network-on-chip (NoC) architecture using real application traffic based on MPEG-2 video decoder. The comparisons are carried out using SystemC cycle-accurate realistic simulations and fault injection experiments employing the fault injection simulator presented in Chapter 3. It is shown that NoC outperforms AMBA in terms of application performance. Furthermore, comparative reliability analyses between the two on-chip communication architectures show that NoC outperforms AMBA in terms of the number of soft errors experienced due to less execution time in NoC.

Chapter 5 outlines the relationship between application-level correctness and system-level power management using voltage scaling technique. Using this relationship, a voltage scaling technique is proposed to generate designs that are optimised in terms of power consumption, while providing acceptable application-level correctness and real-time performance. The effectiveness of the proposed technique is evaluated using an MPEG2 video decoder and synthetic application examples. Using peak signal-to-noise ratio (PSNR) as an application-level correctness metric for MPEG-2 decoder it is shown that the proposed voltage scaling technique can significantly reduce power consumption, while maintaining acceptable application-level correctness and meeting real-time performance constraint. Furthermore, the impact of application task mapping (distribution of application tasks among processing cores in an MPSoC) and architecture allocation (choice of number of processing cores in an MPSoC) is investigated on the trade-offs between the application-level correctness and power consumption. Chapter 6 examines the impact of application task mapping on system reliability in the presence of single-event upsets (SEUs). Based on this study, a novel soft error-aware design optimisation technique is proposed using joint power minimisation and reliability improvement. The power minimisation is carried out using voltage scaling technique and reliability improvement is achieved using application task mapping. The aim is to minimise the number of SEUs experienced by an MPSoC application, while providing low power consumption under a real-time constraint. To evaluate the effectiveness of the proposed design optimisation, different experiments are carried out using a number of applications, including MPEG-2 video decoder and random task graphs. Chapter 7 presents a summary of the thesis along with future and worthy research areas to improve power minimisation and reliability of MPSoCs. The details of the application and simulation models used in the

thesis are described in Appendices A, B and C.

The following peer-reviewed publications have been generated from the research work carried out in this thesis:

Shafik, R. A. and Al-Hashimi, B. M. (In Press) Reliability Analysis of On-Chip Communication Architectures: An MPEG-2 Decoder Case Study. *Elsevier Journal of Embedded Hardware Design (MICPRO).*

Shafik, R. A., Al-Hashimi, B. M. and Chakrabarty, K. (2010) Soft Error-Aware Design Optimization of Low Power and Time-Constrained Embedded Systems. In: *International Conference on Design, Automation and Test in Europe (DATE)*, 2010, Dresden, Germany, pp.1462-1467.

Shafik, R. A., Al-Hashimi, B. M., Kundu, S. and Ejlali, A. (2009) Soft Error-Aware Voltage Scaling Technique for Power Minimization in Application-Specific MPSoC. *ASP Journal of Low Power Electronics (JOLPE)*, vol. 5, no. 2, pp. 145-156.

Shafik, R. A. and Al-Hashimi, B. M. (2009) Comparative Reliability Analysis between AMBA and Network-on-Chip: An MPEG-2 Case Study. In: *International System-on-Chip Conference (SOCC)*, 9-11 September, 2009, Belfast, Northern Ireland, pp. 247-250.

Shafik, R. A., Rosinger, P. and Al-Hashimi, B. M. (2008) SystemC-based Fault Injection Technique with Improved Fault Representation. In: *European Test Symposium (ETS)*, May 25-28, Italy.

Shafik, R. A., Rosinger, P. and Al-Hashimi, B. (2008) SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation. In: *International On-line Test Symposium (IOLTS)*, 7-9 July, 2008, Rhodes, Greece. pp. 99-104.

Shafik, R. A., Rosinger, P. and Al-Hashimi, B. M. (2008) MPEG-based Performance Comparison between Network-on-Chip and AMBA MPSoC. In: *IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 16-18 April, 2008, Bratislava, Slovakia. pp. 98-103.

The following publication is currently under preparation:

Shafik, R. A., Al-Hashimi, B. M. and Chakrabarty, K. (under preparation) Design Optimization of Low Power, Reliable and Time-Constrained Embedded Systems. to be submitted to *IEEE Transactions on Very Large Scale Integration Circuits (TVLSI).*

Chapter 2

Background and Previous Works

This chapter introduces the fundamental concepts involved in this thesis and presents a review of the previous works. The remainder of this chapter is organised as follows. Section 2.1 introduces MPSoC platform and different on-chip communication architectures. Section 2.2 presents the principles of low power design and Section 2.3 introduces reliability and fault injection techniques. Section 2.4 describes the system-level design techniques. Finally, Section 2.5 gives a review of the previous works relevant to the presented research.

2.1 Multiprocessor System-on-Chip

To accommodate more functionality meeting the needs of increased complexity of applications, multiprocessor system-on-chip (MPSoC) is emerging as a popular embedded systems platform [5]. MPSoC contains multiple processing elements on a single piece of silicon, each with an assigned task to define an expected application domain. The inclusion of multiple processing elements in MPSoCs has a number of benefits, including parallel processing, low clock speed and low power consumption [4]. Figure 2.1 shows an MPSoC with two processors, viz. digital signal processor (DSP) and microcontroller unit (MCU), a memory and an application-specific integrated circuit (ASIC). The interconnection of these components within the MPSoC is controlled by on-chip communication architecture. Since on-chip communication architecture defines how inter-component communication takes place, it greatly influences the underlying performance of the system. Currently, there are three major on-chip communication architectures for MPSoCs: point-to-point (P2P), on-chip bus and network-on-chip (NoC). The P2P on-chip communication architecture lays out dedicated interconnects between each communicating component within an MPSoC. Such interconnection results in poor scalability and as such P2P architectures are generally not suited for MPSoCs [6]. The advanced

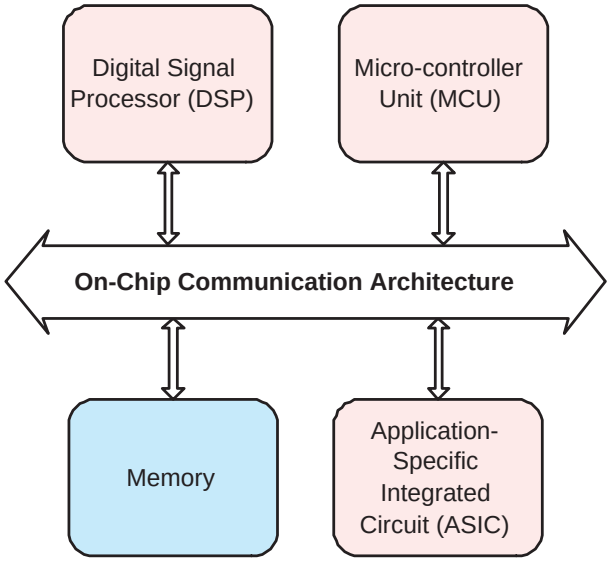


FIGURE 2.1: An example MPSoC showing different components and on-chip communication architecture

microprocessor bus architecture (AMBA) is a dominant, industrial standard on-chip bus architecture for today's MPSoCs, which offers good connectivity and scalability. Network-on-chip is an emerging on-chip communication architecture for the present and future MPSoCs. In the following, AMBA and NoC are briefly introduced.

2.1.1 Advanced Microprocessor Bus Architecture

The advanced microprocessor bus architecture (AMBA) protocol [27] is an open standard, on-chip bus specification that details a strategy for the interconnection and management of functional blocks of a system-on-chip (SoC). AMBA uses a set of signals connected with all other communicating modules, called bus, as the main interconnection unit. Four distinct buses are defined within the AMBA specification [27]: a) the advanced high-performance bus (AHB), b) the advanced system bus (ASB), c) the advanced peripheral bus (APB), and recently specified d) advanced extensible interface (AXI). In this work, AMBA AHB has been chosen as the shared-bus architecture due to its high performance and high clock-frequency [18].

The AMBA AHB bus protocol is designed to be used with a central multiplexor interconnection scheme among various components, including masters (e.g. processing elements, microcontrollers) and slaves (e.g. memory, peripherals). Figure 2.2 shows a block diagram of the interconnection scheme used in AMBA. Using such scheme all bus masters drive out the address (HADDR) and control signals (Read: RD, Write: WR) to the address/control multiplexor indicating the transfer they wish to perform. The arbiter determines which master has its address from address/control multiplexor and routes the control signals (Read: RD, Write: WR) to all slaves. Each of the master

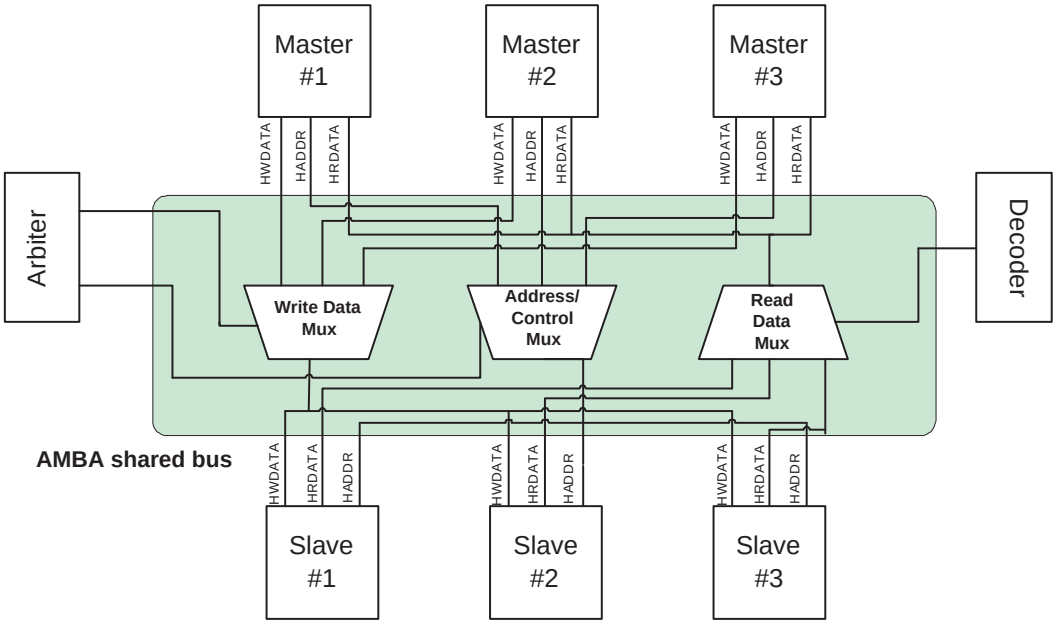


FIGURE 2.2: Block diagram of AMBA AHB central multiplexing scheme connecting arbiter, decoder, masters and slaves

HWDATA interface (for write data interface) is connected to the slaves through AMBA write data multiplexor. Similarly, the HRDATA interfaces (for read data interface) from slaves is connected to the masters through a read data multiplexor. A central decoder is required to control the read data multiplexor and select the appropriate signals from the slave that is involved in the transfer (Figure 2.2).

Figure 2.3 shows how a data transfer takes place through AMBA AHB. As can be seen, before an AMBA AHB transfer can commence the bus master must be granted access to the bus (Figure 2.3). This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus. A granted bus master then starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the type of transfer: burst, incrementing burst or wrapped burst. A write data bus (HWDATA) is used to move data from the master to a slave, while a read data bus (HRDATA) is used to move data from a slave to the master. Every transfer consists of an address and control cycle and one or more cycles for the data. The address cycle cannot be extended and therefore all slaves must sample the address during this time. The data cycle, however, can be extended using the HREADY signal. When HREADY is LOW, wait states are inserted into the transfer to allow extra time for the slave to provide or sample data. The address cycle of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus and allows for high performance operation, while still providing adequate time for a slave to provide the response to a transfer.

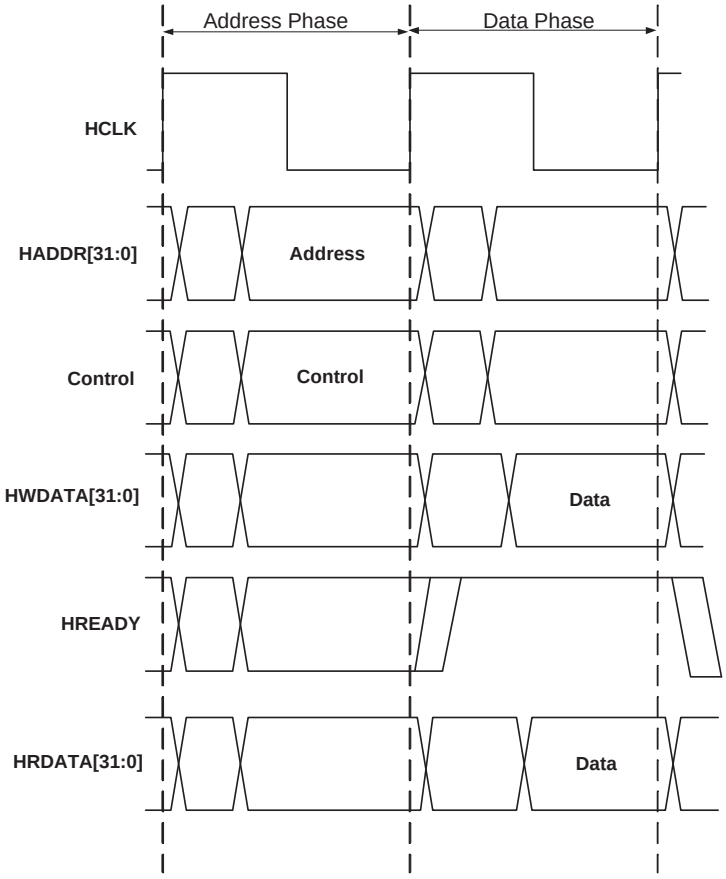


FIGURE 2.3: A simple AMBA AHB data transfer without waiting states

Depending on how the components are connected in AMBA, a number of different on-chip bus architectures are possible [28]. Single layer shared-bus AMBA, where only one master can initiate and carry out communication at a given time, is a dominant on-chip bus architecture [29, 30]. Such architecture has benefits, such as scalability and high clock frequency [31] and is used in this work.

2.1.2 Network-on-Chip

Network-on-Chip (NoC) is an emerging on-chip communication architecture enabling integration of large number of processing elements on a single chip. NoCs offer modular structure and uses packets for on-chip communication. Due to high level of modularity, NoCs have advantages of high scalability and performance at the expense of silicon area and complexity [32, 33, 34]. A great deal of research works have already been carried out to explore efficient and reliable NoC topologies and routing techniques [15, 33].

An NoC is made up of the following components:

Processing Element (PE) is responsible for computation. Examples of PEs are microprocessor and digital signal processor (DSP).

Network Interface (NI) is generally attached to a PE and is responsible for packet-based communication. NI adds communication related headers and tails to generate packets for outgoing data from PEs or discards the headers and tails from a packet as it enters PEs.

Switch is responsible for communication related tasks within NoC and is attached to one or more PEs through the NIs. Switch contains multiple buffers to store packets from various directions and arbiters to select single output channel from a number of outgoing packets. The selection of outgoing packet and selection of communication links are controlled by a set of routines defined by a routing technique.

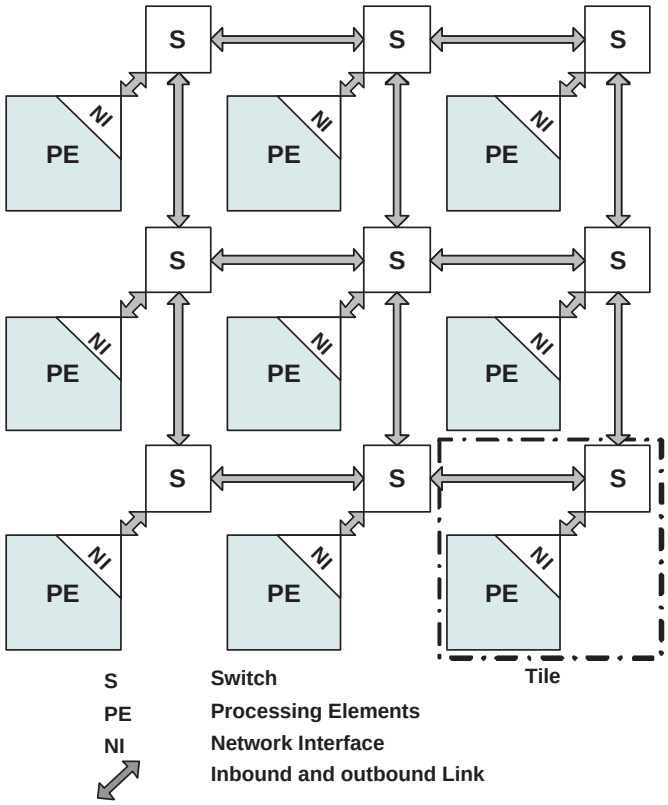


FIGURE 2.4: A (3×3) NoC architecture with CLICHE or mesh-based topology

NoC offers a high degree of freedom with interconnection of its components and underlying communication and routing algorithms. In the following, NoC topology, the packet routing and communication techniques used in this thesis are briefly introduced.

NoC topologies define the way the PEs and NIs are connected with the switches and have direct influence on the performance of the system [35, 36]. Mesh-based interconnect architecture, also called CLICHE (chip-Level integration of communicating heterogeneous elements), is a popular NoC architecture, proposed by [37]. Figure 2.4 shows a simple (3×3) mesh-based NoC. As shown, every switch in such topology is connected to a

specific PE and the number of switches is equal to the number of PEs. All switches are connected to the four closest switches and the target resource block, except those on the edge of the layout. Due to its regular structure, mesh-based topology has good scalability with simple switches [38, 39, 40]. The simplicity mesh layout allows for the division of the chip into processing or resource regions. The smallest of such region is a tile formed of single PE, NI and switch (Figure 2.4). Although mesh-based NoC is very well known for its scalability and performance, it incurs higher wire overhead (approximately 12% higher than AMBA [17]) due to large number of switches and wires. A large proportion of these switches is taken up by the processing logic of the router, which estimated to be approximately between 2.0% [40] to 6.6% [38] of the total chip area.

NoC routing algorithms are crucial as they determine the paths the packets take during inter component communication. A good routing algorithm reduces the latency of the network by minimising the number of hops that are required for packets to reach their destination. For packet switched NoCs, routing techniques and algorithms have direct impact on the communication and system performance [41]. NoC routing algorithms are classified as either deterministic or adaptive. The deterministic routing algorithms choose a route without considering any information about the network's present condition, resulting in less design complexity [42]. Adaptive routing algorithms use the state of the network like the traffic status of a node or link and the status of buffers for network resources to adapt to a new routing path [43].

The XY routing, proposed by [44], is a popular routing technique for mesh-based NoCs. In this routing, the position of the mesh nodes and their nested network components is described by coordinates: the X-coordinate for the horizontal and the Y-coordinate for the vertical position. A packet will be routed to the correct X-direction first and then in Y-direction. XY routing is high-performance and simple to implement [44]. Figure 2.5 shows XY routing for packet travelling from from node (2,2) to node (0,0) in a (3×3) mesh-based NoC. The routing takes the path in the X-direction first and then Y-direction, involving the nodes as follows: $(2,2) \longrightarrow (2,1) \longrightarrow (2,0) \longrightarrow (1,0) \longrightarrow (0,0)$ (Figure 2.5). Among other routing techniques, source-based and odd-even turn model [45] are examples of deterministic and adaptive routing techniques. In source-based routing technique, the direction of packet travel is specified in the packet header, making the switch design simple. In odd-even turn model for routing the path of packet travel is determined adaptively. Two main restrictions are imposed in such routing technique to avoid deadlock of packets. Firstly, no packet is permitted to do east-north or north-west turn at a node if it is located in an even or odd column (column 0 is odd, column 1 is even and so on). Secondly, no packet is permitted to do east-south turn or south-west turn at a node if it is located in an even or odd column [45]. Figure 2.6 shows an example of odd-even routing for packet travelling from from node (0,2) to node (1,0) in a (3×3) mesh-based NoC. As can be seen, the packet starts travelling from an odd

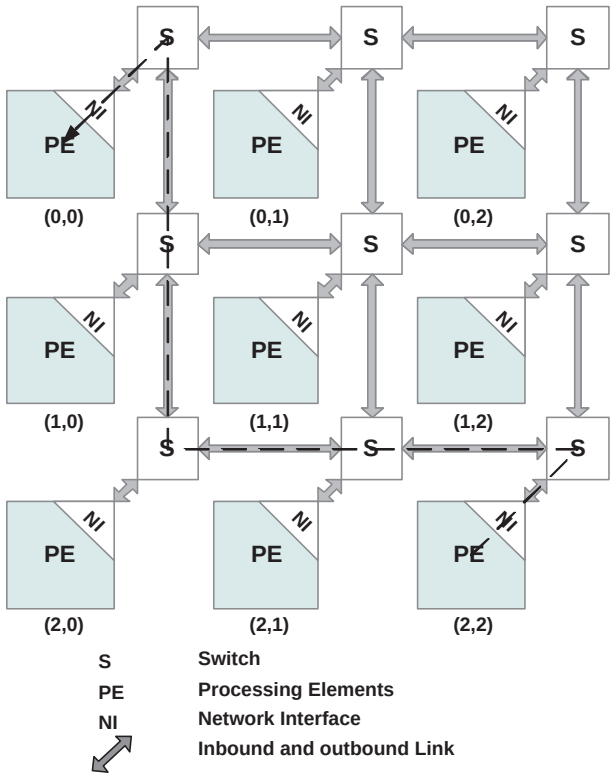


FIGURE 2.5: An example of XY routing between node (2,2) to node (0,0) in a (3×3) mesh-based NoC

column (column 2) from node (0,2) and travels to node (1,2). After reaching node (1,2) the packet can take a south-west (first south and then west) turn and follow the nodes (2,2) and (2,1). After reading node (2,1) (which is in an even column), the packet can now take a west-north turn and finally reach the destination node (1,0). Note that unlike deterministic routing techniques, such as XY routing or source-based routing, odd-even turn model can follow any permitted path from a node depending on the node status.

Packet structure is a key design consideration in NoC. An NoC packet generally consists of header, payload and tail as shown in Figure 2.7. The header contains useful information regarding the packet communication, while the payload contains the actual data that is to be communicated. The tail marks the end of packet. The header information added by the source network interface at the front end of the packet generally contains various information to enable packet-based communication, including routing and communication history. Depending on the routing algorithm used, packet structure can be different and packet headers can be modified by the intermediate switches.

Depending on how a packet is communicated from source to destination, NoC packet communication can be classified in three techniques: store and forward, virtual cut through and wormhole. Figure 2.8 shows the three different packet communication techniques. The store and forward technique requires the whole packet to be received and stored by the switch before it can be sent to the next node (Figure 2.8(a)). The vir-

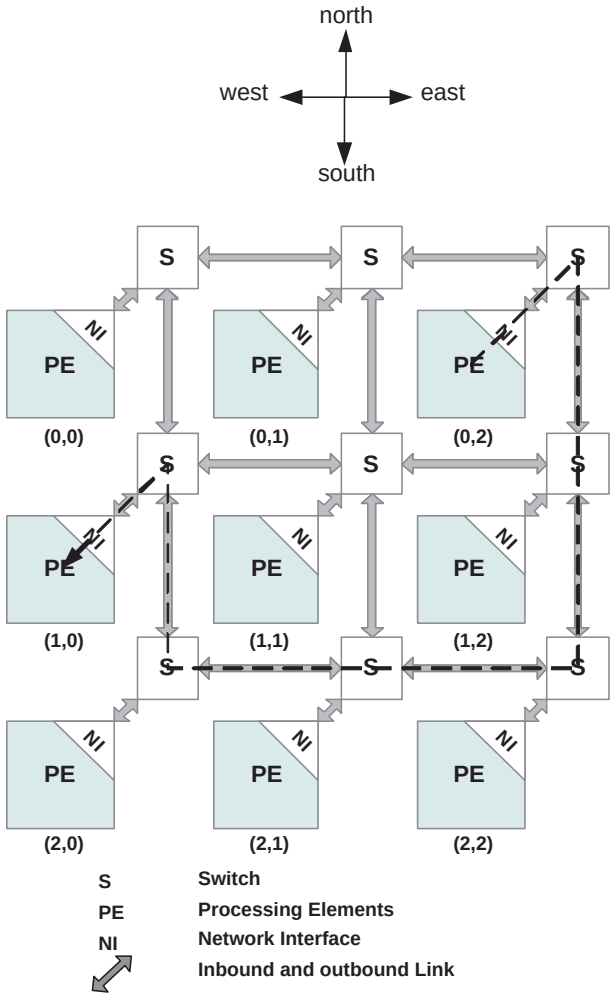


FIGURE 2.6: An example of odd-even routing between node (0,2) to node (1,0) in a (3×3) mesh-based NoC

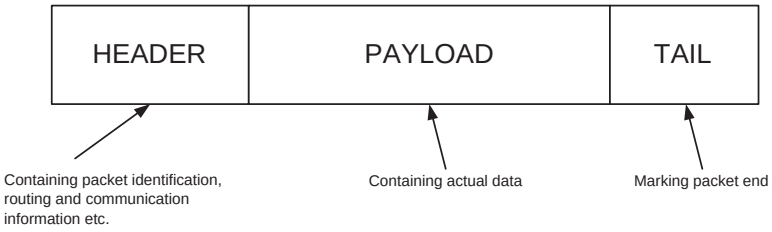


FIGURE 2.7: A general structure of NoC packet with different components

tual cut through technique is similar to store and forward, except that the switches can start sending packets before the entire packet is received (Figure 2.8(b)). The wormhole technique is a finer grain technique and requires the packet to be split in flow control units, called flits (Figure 2.8(c)). The idea is to split a large packet communication into small modular flits, which has the advantage of reduced wire length. Each flit is considered as the unit of communication between switches and interconnecting elements. All flits contained within a packet follows the first flit in subsequent communication cycles.

Both store and forward and virtual cut through requires the switches to have enough

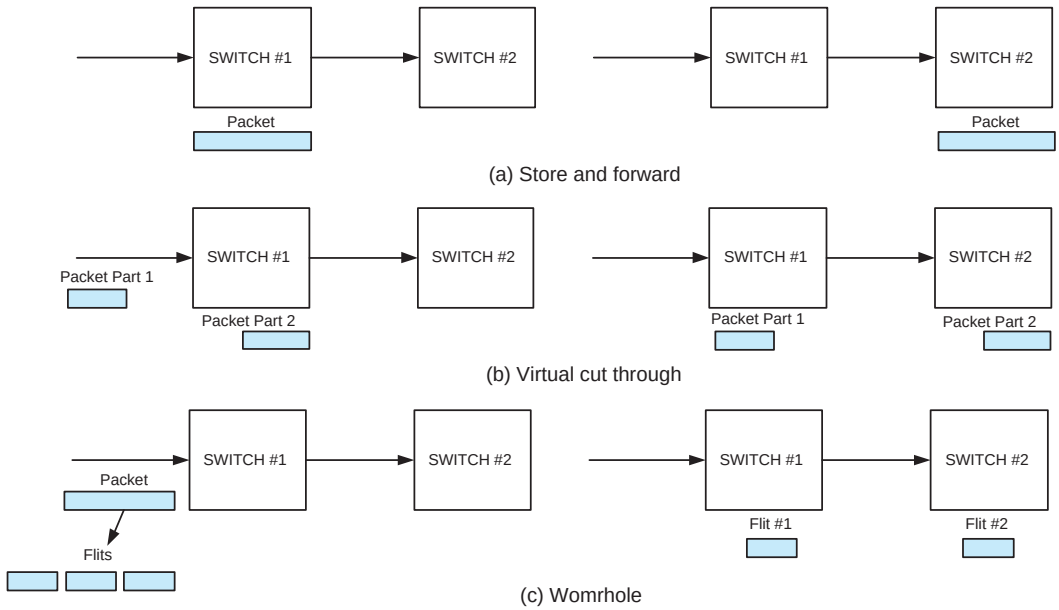


FIGURE 2.8: Different packet communication techniques: (a) store and forward, (b) virtual cut through, and (c) wormhole packet communication technique

buffer space to hold and entire packet. If the packet size is larger, this can make the switches have considerable overheads. On the other hand, although wormhole requires less buffer space but as every flit needs to be acknowledged, it creates considerable traffic into the NoC [41]. Also, due to reduced wire in communication channels in the switch, the switch area is smaller for such communication technique.

2.2 Low Power Design

The power dissipated by processing elements (PEs) of an MPSoC is given by the sum of the leakage power, P_{leak}, and dynamic power, P_{dyn}, as [46]

$$P_{PE} = P_{leak} + P_{dyn}. \tag{2.1}$$

The dynamic power, P_{dyn}, in (2.1) is caused by switching activity of the PE, while the leakage power, P_{leak}, is the power present when no switching activity takes place in the PE. The dynamic power, P_{dyn}, is given by

$$P_{dyn} = \alpha \; C_L \; V_{dd}^2 \; f, \tag{2.2}$$

where C_L is the processor load capacitance per cycle (generally constant for a given PE), V_{dd} is the supply voltage, f is the operating frequency and α is the processor activity factor (the ratio of switching activity over a given time). From (2.2) it is evident that the

most effective means of lowering P_{dyn} is to reduce V_{dd}. However, lowering V_{dd} increases circuit propagation delay [47]. This delay restricts the circuit operating frequency in a PE and hence it is required to lower the clock frequency to tolerate the propagation delay [47]. Dynamic voltage scaling (DVS) is an effective power minimisation technique that reduces power through lowering V_{dd} and f during runtime. The main working principle of DVS is to lower V_{dd} and f during slack times caused by the idle period between computational tasks due to early completion the previous computational task or late starting of the next computational task [48]. Over the last decade, power minimisation using DVS-enabled MPSoCs has been extensively investigated considering its effects on system performance due to lowering of operating frequency [5, 49, 50, 51, 52].

Dynamic power management (DPM) is another effective technique to reduce power consumption [53]. The main strategy employed in DPM is to control the operational times of supply voltages in system components. For example, power supply can be shut down for components within an MPSoC when they are idle and can be switched on when they are operational. However, shutting down power for these components results in delay in waking them up to their fully operation condition. Hence, DPM technique needs to take into consideration this delay effect to achieve power minimisation without compromising the system performance [54]. Often, today's MPSoCs include both DVS and DPM techniques to minimise power. In this work, power minimisation is achieved out using voltage scaling technique as it is widely employed for power minimisation of MPSoCs [47, 48, 55]. Reviews of research works related to voltage scaling and its effects on system performance and reliability is presented in Section 2.5.

2.3 Reliability and Fault Injection

An emerging challenge in today's electronic system design is reliability of the system when it is subjected to different errors or faults. The existence of these errors highlighting the impact of technology scaling and operating environments has been investigated in a number of academic and industrial studies, such as [10, 11, 12, 56]. To determine how reliability is affected by the presence of these errors, fault simulation has been a prevalent technique over the past few years [57]. Fault simulation describes the technique of fault injection in a prototype implementation through a simulation model to observe the behaviour of the system in the presence of faults [58]. Different fault injection techniques using simulated fault injection are briefly introduced in Section 3.1, Chapter 3.

Faults in electronic systems can be classified in two types: permanent and transient. Permanent faults are related to irreversible physical defects in the circuit. These defects can be produced during manufacturing process or in normal operation. In simulated fault injection techniques such faults are made persistent throughout the observation time. Stuck-at faults are popular permanent fault model used in fault injection experi-

ments [57]. Transient faults, also known as soft errors, can appear during the operation of a circuit. Unlike permanent faults, transient faults do not represent a physical defect in the circuit. Such faults take place when a single ionising radiation event produces a burst of hole-electron pairs in a transistor that is large enough to cause the circuit to change state. Bit-flip or single-event upset (SEU) is the most popular transient fault model used in fault injection techniques [59]. Such faults generally take place at the storage elements, like registers. Among other transient fault types, indetermination and delay faults are also used in fault injection experiments [57]. Indetermination type faults are caused by increased leakage current, which perturbs the original state ('0' or '1' values) of logic gates to unknown state ('X'). The delay type faults are transient delay experienced by logic gates due to change of physical properties, such as capacitance and resistance.

Probabilistic simulation is an effective and popular technique for transient fault injection, which uses statistical information regarding locations and rates associated with faults [60, 61]. For SEU-based fault injection, Poisson's distribution is generally used to identify the fault locations within the registers [62, 63], while exponential distribution is used for determining the timing of fault injection [64]. Different parameters have been reported to date showing methods of evaluating the rate of faults occurring in the device under test (DUT). These parameters include:

Fault Density is the measure of number of faults found in a device per unit of data. For memories, this is generally expressed as the number of faults per megabyte or gigabyte data. This parameter is used for permanent faults or defects only [7].

Failures in time (FIT) is the rate at which the failures or faults take place per unit of time in an electronic component. It is generally expressed in unit of number of fault per million of operating hours (fault / 10^9 hours).

Mean time-to-failure (MTTF) is an estimate of the mean time expected until the first fault occurs in a component of an electronic system. MTTF is a statistical value and is meant to be the mean over a long period of time and large number of units. It is usually expressed in unit of millions of hours. For constant failure rate systems, MTTF is the inverse of FIT [65].

Mean time-between-failures (MTBF) is described as the time elapsed before a component in an electronic system experiences another fault. Unlike MTTF, the time elapsed in MTBF includes the time required to recover from a fault.

Soft error-rate (SER) is the rate at which the errors take place per unit time and per unit data. It is generally used for soft errors and is expressed as number of soft errors per bit per cycle.

In this research, SEU-based probabilistic fault injection technique is carried out using soft error rate (SER) as it is commonly used for evaluating reliability in the presence of SEUs [65].

2.4 System-Level Modelling and Design

MPSoC design is complex due to feature richness, application complexity and short time-to-market requirements [66]. System-level modelling and design address these challenges in MPSoC design using modular design approach at a higher-level of abstraction. Over the years, a number of system-level modelling and design tools and techniques have been developed for academic and industrial research and developments, such as [21, 26, 67]. In this work, SystemC is used for system-level modelling and simulation of MPSoC (Chapters 4, 5 and 6). The different concepts involved in system-level modelling and design used in this research are briefly described next.

2.4.1 System-level Modelling and Simulation using SystemC

SystemC, ratified as IEEE standard 1666-2005 [68], is a hardware description language (HDL), which extends standard C/C++ with the use of HDL-specific libraries. It has been developed by a group of companies forming the Open SystemC Initiative (OSCI) [68]. Like other HDLs, such as VHDL and Verilog, SystemC provides a large set of hardware data types along with their array-based implementations. It abstracts design component as re-usable modules and incorporates timing information, concurrent process modelling and process sensitivity. Furthermore, it provides with a rich set of communication classes, such as channels, which can be used to model complex communication behaviour between SystemC modules easily.

To demonstrate the organisation and design specification in SystemC, Figure 2.9 shows an example SystemC hardware description of 1-bit adder. The description of 1-bit adder is organised in hierarchical modules: basic module, testbench module and top-level module (Figure 2.9). The basic module (which is the adder module, *SC_MODULE(Adder)*) incorporates the desired functionality of the system and contains the input and output ports (input ports: *sc_in*<..>, output ports: *sc_out*<..>), functionality of the module, *add(..)*, and the constructor for the module, *SC_CTOR(..)*. The input port signals *A*, *B* and *Cin* (of logic type) provide interface for 1-bit adder inputs with carry-in and the output port signals *sum* and *Cout* (of logic type) provide the sum and carry-out outputs. The actual functionality of adder is described in *add(..)*. The constructor creates the module instance and registers the *add(..)* functionality as a SystemC process using *SC_METHOD* making it sensitive to the input port signals. Such *SC_METHOD* based process registration enables *add(..)* functionality to be executed recurrently when any

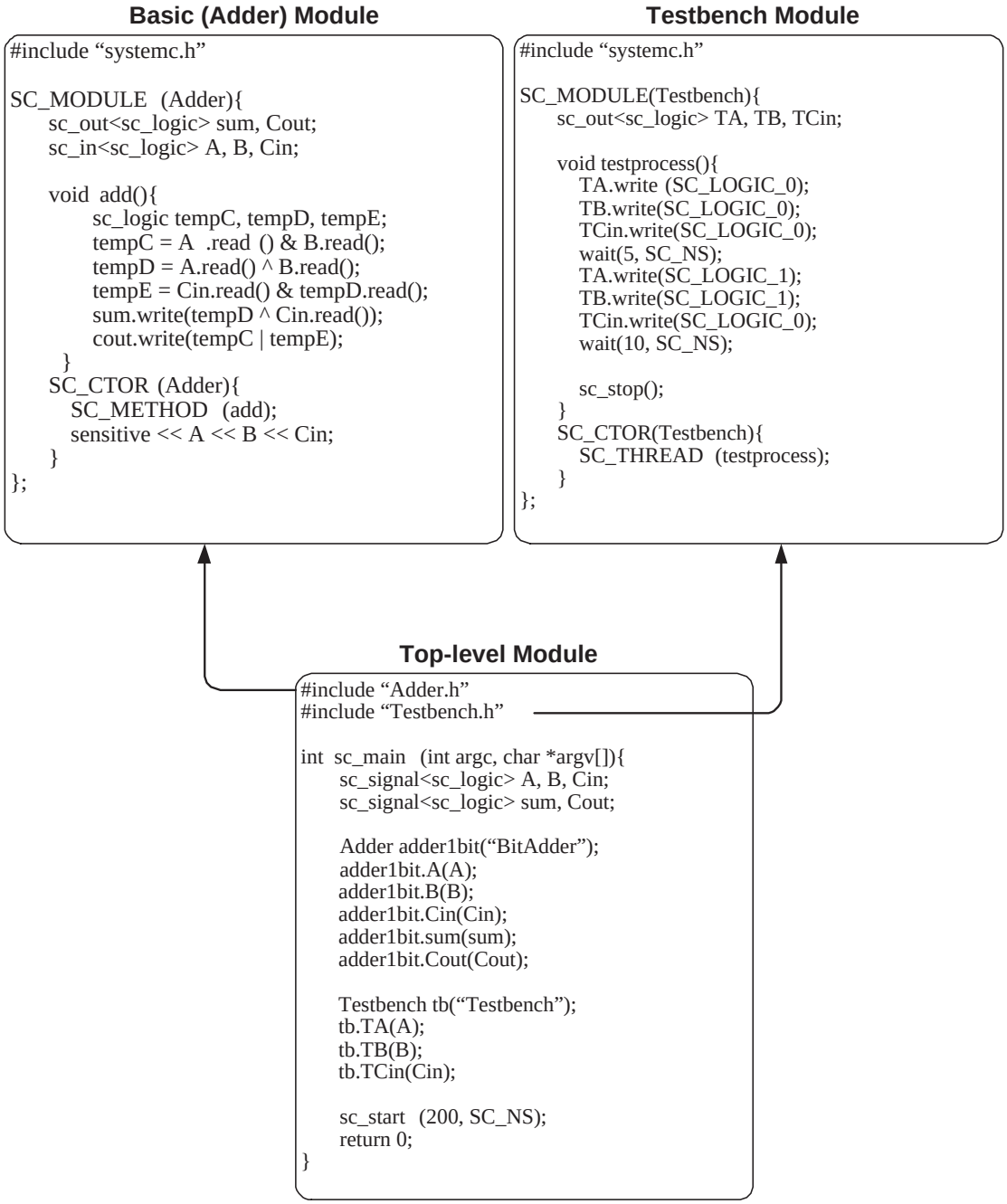


FIGURE 2.9: An example SystemC model of a 1-bit adder

of the sensitivity signals change (*Adder* module, Figure 2.9). The testbench module (*SC_MODULE(Testbench)*) incorporates validation of the adder functionality through different sequences in output port signals: *TA*, *TB* and *TCin* (of logic type). The validation is carried out through SystemC process called *testprocess(..)*, which is registered in the testbench constructor, *SC_CTOR(..)*, using *SC_THREAD*. Such *SC_THREAD* based process registration enables *testprocess(..)* to be executed once (*Testbench* module, Figure 2.9). The top-level module integrates the basic module (*Adder*) and testbench module (*Testbench*) together connecting the outputs of the testbench module to

the inputs of the adder module. The top-level module also issues simulation specific initialisations that are required, such as *sc_start(..)* to start SystemC simulations (top-level module, Figure 2.9). A detailed language specification of SystemC can be found in [69].

2.4.2 Hardware/Software Co-design

Hardware/software (HW/SW) co-design describes the techniques for simultaneous design and synthesis of both hardware- and software-based implementations [70]. Such design approach extracts maximum benefit of both hardware- and software-based design and reduces time-to-market greatly [71, 72]. Figure 2.10 shows a typical design flow used in HW/SW co-design. In the following the major HW/SW co-design steps are briefly introduced. The impact of the design steps on power minimisation and reliability are considered in Chapters 4, 5 and 6.

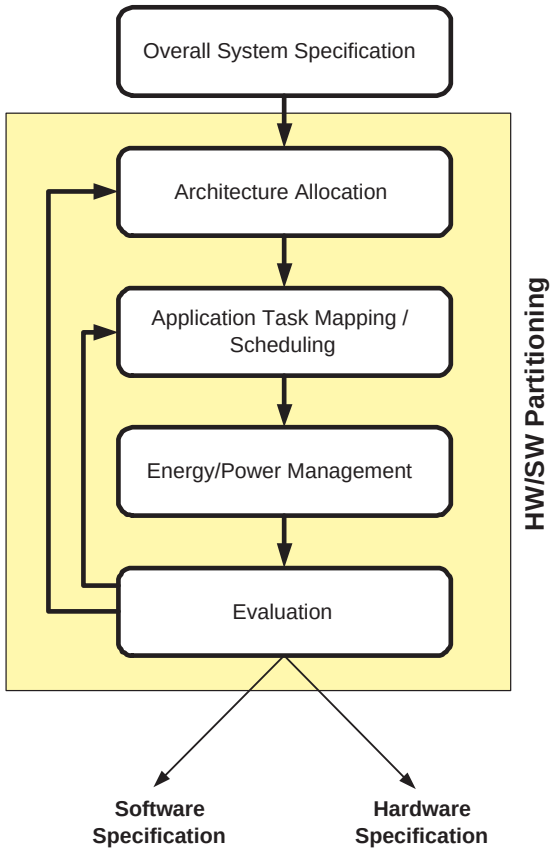


FIGURE 2.10: Flowchart of hardware/software co-design

2.4.2.1 Architecture Allocation

Architecture allocation deals with allocation of different design components for a target MPSoC [48]. For example, architecture allocation in an MPSoC may include selection of number and types of processors, memories and interconnections. The overall goal of the architecture allocation process is to identify the most suitable architecture, which would provide the best performance and cost under given constraints. In this thesis, architecture allocation is referred to as the allocation of number of processing cores in the MPSoC architecture.

2.4.2.2 Application Task Mapping and Scheduling

Following the architecture allocation, application task mapping describes the process of distributing the computation and communication tasks among the allocated processing and communicating elements. The mapping considerations may include different performance and cost involved with each task. The mapping process explicitly determines the implementation related issues in hardware and software. Determining a good mapping is of crucial importance as inappropriately mapped tasks for a given allocated architecture may under-utilise its performance and increase the system costs [73]. Task scheduling generally follows task mapping, which defines the order of execution of the different tasks, such that timing constraints are satisfied. Task scheduling affects the parallelism among different tasks and performance of the system [48, 74].

Architecture allocation, application task mapping and task scheduling have direct impact on performance and power consumption of MPSoC [48, 75]. Further power minimisation can be achieved using dynamic voltage scaling (DVS) technique and dynamic power management using slack time (Section 2.2). The power management is then followed by system evaluation, which may repeat the previous design steps to achieve optimised HW/SW partitioning and design. Over the years a number of HW/SW co-design tools have been proposed that can carry out the above tasks, such as COSYN [71], POLIS [76], MPARM [21]. In Chapter 5 MPARM [21] has been used as it enables high-level description of the design using C/C++ for the software implementation and SystemC for hardware implementation. An introduction to MPARM is presented in Section B.2, Appendix B.

2.5 Previous Works

Significant progress has been made in the research and development of efficient on-chip communication architecture of MPSoC. Advanced microprocessor bus architecture (AMBA) is today's dominant on-chip communication architecture [18] (Section 2.1.1).

With increased complexity of applications and shorter time-to-market demands, the design of on-chip communication architecture is being expedited further towards more modular and scalable architectures, such as network-on-chip (NoC) [77, 78] (Section 2.1.2). A number of studies have discussed the benefits of NoCs as a future MPSoC platform. In [79] it was shown that shared-bus will not meet the performance requirements of tomorrow's systems. It demonstrates that network-based systems would outperform shared-bus based systems with better throughput and lower latency. Further justifications were presented in [38] showing that using an NoC-based on-chip communication architecture has advantages of structure, performance and modularity. Since network channels are shared across all connecting components, the bandwidth is enhanced but this gives NoC an area overhead of 6.6% [38]. It also shows that unlike in point-to-point (P2P) architecture, the wiring in NoC is not dedicated to connecting components only, which also enhances utilisation. Similar in-depth and intuitive analyses of NoC concepts and arguments are also presented in [77, 80, 81]. In [19] an overview of the research works carried in the development of NoC is presented. Also, a set of case studies with industrial and academic prototyping and implementation has been summarised in [19] showing the NoC implementations: ÆTHEREAL by Philips [82], NOSTRUM [83, 84], SPIN [79], CHAIN [85], MANGO [86] and xPipes [87]. Further review of related research works related to comparison between NoC and AMBA is presented in Section 4.1, Chapter 4.

Power reduction is a major consideration in MPSoC design to extend battery life. Dynamic voltage scaling (DVS) is an effective power minimisation technique that has received considerable attention over the last decade [5, 48, 49, 50, 51, 52, 54, 55] (Section 2.2). However, power minimisation using such technique has detrimental effects on the system performance and reliability. This is because lowering of the operational frequency degrades the performance of the system and lowering voltage to achieve power minimisation exponentially increases the number of soft errors experienced. This has been extensively investigated in a number of studies [62, 63, 88, 89, 90, 91].

To mitigate the effect of soft errors in low power systems, different techniques have been employed over the last few years [48]. Hardware redundancy technique, such as [89, 92], is an effective technique, which employs extra hardware and incorporates voting from multiple outputs to a single output to mitigate the effect of soft errors. Due to usage of extra hardware resources such techniques incur area and power overheads. A number of publications have reported this conflict between redundant resources and power consumption for fault tolerant designs, for example [89]. Time and information redundancy techniques are other effective fault tolerance techniques. However, several studies reported that these techniques cause overheads in terms of computation and communication performance. For example, in [93, 94] the trade-off between power minimisation and performance using time redundancy technique is shown, while in [62] the trade-off between communication overhead and power consumption using information-redundancy is presented. An alternative technique is to employ task re-execution or replication as

shown in [13, 95]. Using this technique fault tolerance is achieved through task replication and re-execution during idle/slack times and power minimisation is achieved using task scheduling. The overhead for such technique is generally much lower than redundancy technique. For example, in [93] it is demonstrated that the fault tolerance can be improved without any impact on the execution time by utilising idle processors for duplicating some of the computations of the active processors. Application check-pointing is another effective technique. The fault tolerance using such technique is achieved by selectively repeating part of an application during slack times to realise fault tolerant design. However, the fault tolerance using this technique is achieved at the cost of high complexity of application design. Examples of effective application check-pointing techniques highlighting such increased cost are, adaptive and non-uniform online application check-pointing proposed in [96, 97], offline application check-pointing shown in [50]. A number of other techniques have also been proposed in past few years showing the combination of different fault tolerance techniques to extract maximum benefit in terms of fault tolerance and low power consumption, such as [74, 98, 99]. Further review of research works related to fault tolerance and reliability is presented in Section 6.1, Chapter 6.

Recently, to ease fault tolerance requirements for DVS-enabled systems, different techniques have been proposed showing the impact of faults at application-level. Since the impact of faults are investigated at application-level rather than architectural-level, a large number of faults get masked enabling low-cost fault tolerance technique. In [7, 8] it has been shown that multimedia applications are capable of tolerating a high rate of faults. A similar study has been reported in [24] showing that soft errors at architectural-level do not always lead to faults at application level, leading to the concept of application-level correctness. Exploiting the relaxed requirements of application-level correctness, the authors of [24] demonstrated that effective and fault-tolerant applications can be designed at low-cost. Further review of research works related to the evaluation of the impact of faults is presented in Section 5.1, Chapter 5.

2.6 Concluding Remarks

This chapter has introduced the fundamental concepts related to the thesis. Different MPSoC on-chip communication architectures, such as AMBA and NoC, have been described. Low power design techniques using dynamic voltage scaling (DVS) and dynamic power management (DPM) have been discussed and concepts related to reliability and fault injection have been outlined. Also, overview of different system-level design techniques and SystemC-based modelling and design specification have been given. Furthermore, a review of previous research works relevant to this thesis has been presented.

Chapter 3

SystemC Fault Injection Simulator

With the ongoing trend of technology scaling, reliability is an emerging issue in the design of electronic systems, particularly due to transient single-event upsets (SEUs) and manufacturing permanent faults [100] (see Section 2.3, Chapter 2 for further details on reliability). Hence, an important aspect in the design of electronic systems is to validate the feasibility of fault tolerance at a high level of design abstraction to reduce the system re-design cost. Such validation requires studying how different faults affect the system functionality and behaviour. Simulated fault injection is an effective technique often used to facilitate such studies [58]. Desirable features of an effective fault simulator include low intrusion into original design description, low simulation time and implementation of various type of fault injection. SystemC is a system-level design and specification language with potential improvements in design productivity by allowing the designer to operate at different levels of abstraction. Currently, there is a lack of a SystemC fault injection simulator with the desirable features of an effective fault simulator. In this chapter, a SystemC-based prototype fault injection simulator is presented employing a novel fault injection technique. The simulator benefits from simplicity, minimum design intrusion and high fault representation compared to other recently reported SystemC fault injection techniques.

This chapter is organised as follows. Section 3.1 reviews existing fault injection techniques. Section 3.2 describes the new fault injection technique implemented on a prototype simulator. Section 3.3 demonstrates the effectiveness of the simulator and compares with recently reported fault injection techniques. Section 3.4 considers MPEG-2 video decoder setup as a case study using the proposed fault injection simulator. Finally, Section 3.5 concludes the chapter. The fault injection simulator is also used in Chapters 4, 5 and 6 to analyse the reliability in the presence of soft errors.

3.1 Related Works

To date a number of fault injection simulators have been proposed employing different design description languages. For example, a VHDL-based fault injection simulator, called MEFISTO, is presented in [101] showing fault injection through signal or variable manipulation. Similar fault injection simulators using signal or variable manipulation for fault injection are also shown in [102, 103]. The simulators in [102, 103] employ automatic design modification through VHDL-based controller. Such design modification is demonstrated to be useful for speeding up the system re-design process and enable effective fault injection in behavioural design description of the device under test (DUT). Another VHDL-based fault injection simulator, called VERIFY, is proposed in [104]. The simulator injects faults in the DUT by flipping bits within the registers during simulation. A Verilog-based fault injection simulator, called INJECT, is shown in [105] using mutated design description to enable fault injection. Examples of other effective fault injection simulators employing different techniques are DEPEND [106], GOOFI [107] and RIEFLE [108].

SystemC is a design description language that allows the designer to operate at different levels of abstraction [68]. Currently a number of SystemC fault injection simulators have been proposed employing different fault injection techniques, such as saboteurs [60], mutants [109] or simulation command based approach [110]. Apart from simulation techniques, emulation approach for fault injection have also been proposed, such as [111]. In the following, the existing fault injection techniques in the context of SystemC are discussed. These techniques are compared with the novel fault injection technique implemented on a prototype simulator in Section 3.3.

3.1.1 Saboteurs

A saboteur is a special component added to the original design description of the device under test (DUT). The purpose of this component is to alter the value or the timing characteristics of one or more signals in the original design description, when a fault is injected [112]. Due to its simplicity, it is a popular fault injection technique [59]. Depending on how saboteurs are connected with the signals in the original design description, it can be serial simple, serial complex or parallel as shown in Figure 3.1 ('O' - signal driver, 'S' - saboteur and 'I' - signal receiver). In serial configuration, saboteur is inserted between a signal driver and its receiver (Figure 3.1(a)). In serial complex configuration, saboteur is inserted between two different sets of signal drivers and signal receivers to simulate cross-talk based faults (Figure 3.1(b)). A parallel configuration is used when a set of drivers and receivers have a common resolved signal (Figure 3.1(c)).

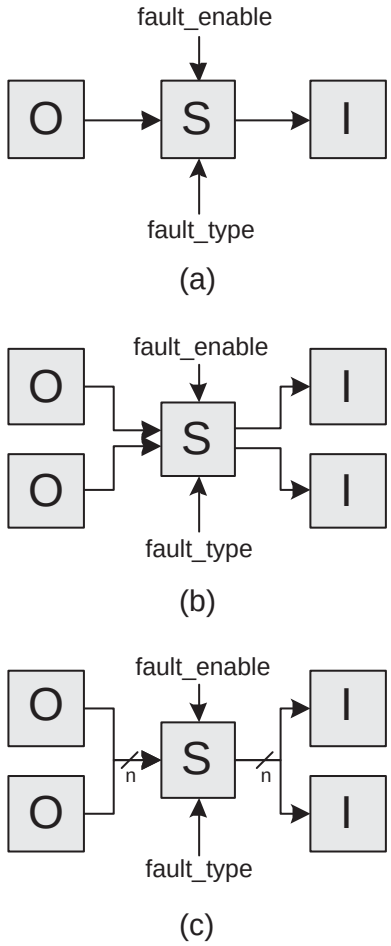


FIGURE 3.1: Different types of saboteurs (unidirectional only): (a) Serial simple saboteur, (b) Serial complex saboteur, and (c) Parallel saboteur

Figure 3.2 shows an example SystemC design of a serial simple saboteur (Figure 3.1(a)). To make the saboteur compatible with different signal types, a template class is used. As can be seen, the template class consists of three major parts: the saboteur input/output, saboteur constructor and the actual fault injection functionality (Figure 3.2). The saboteur input (*sab_in*) drives the signal for which the fault injection is being enabled and the saboteur output (*sab_out*) receives the signal value after fault injection (lines 4-5). The fault injection is controlled by the fault enable signal, *fault_enable*, and the fault type is handled by *fault_type* (lines 6-7). The saboteur constructor creates an object of the saboteur registering the main fault injection functionality, *insert_fault*, as an *SC_METHOD* process (see Section 2.4, Chapter 2 for example of how processes are registered within SystemC modules). The fault injection functionality, called *insert_fault(..)*, is made sensitive to saboteur driver, *sab_in* in the constructor such that any change in the *sab_in* causes *insert_fault* functionality to be executed (lines 9-12). Within the *insert_fault(..)*, when the *fault_enable* signal is enabled, the driver signal, *sab_in*, is altered by inserting a random fault into it (lines 17-21). However when the *fault_enable* signal is disabled, no faults are injected and the original signal remains fault-free. The variable *data* acts

```
//Saboteur.h
1:  template <typename T>
2:  class Saboteur: public sc_module{
3:  public:
4:    sc_in<T> sab_in;
5:    sc_out<T> sab_out;
6:    sc_in<bool> fault_enable;
7:    sc_in<sc_uint<2> > fault_type;
8:
9:  SC_CTOR(Saboteur){
10:   SC_METHOD(insert_fault);
11:   sensitive << sab_in;
12: }
13:
14: void insert_fault(){
15:   if(sab_in.event(){
16:     data = sab_in.read();
17:     if(fault_enable.read() == 1){
18:       data = sab_in.read() + fault;
19:     }
20:     sab_out.write(data);
21:   }
22: }
23:
24: private:
25:  T data;
26: };
```

Saboteur I/O and fault information

Saboteur constructor

Fault injection

FIGURE 3.2: An example SystemC template of a serial simple saboteur (Figure 3.1(a))

as a local variable to store the faulty or fault-free data, which is written to the signal receiver, *sab_out* to reflect the fault injection (lines 20). For simple design description, using such saboteur (Figure 3.2) is an effective technique for fault injection [110].

Saboteurs can only manipulate signals to enable fault injection (Figure 3.2). Hence, saboteurs are not suitable for fault injection in behavioural models of the DUT, which generally include variable registers apart from signal registers [57]. A saboteur-based SystemC fault injection simulator is presented in [60]. The fault injection is enabled through manipulation of signals in the DUT using similar saboteur templates similar as shown in Figure 3.2. In [113] a SystemC-based fault injection technique is proposed using saboteur-based controllers to flip bits during data transfer of a system. A similar system-level fault injection technique using SystemC is presented with distributed saboteur-based fault injection controllers in [114]. The fault injection controllers in [113, 114] define the interface between input and output of the set of signals and enable fault injection through external inputs.

3.1.2 Mutants

A mutant is a design component that replaces the original design component to enable fault injection. It works as the original component when inactive and works as a

faulty component when active. Mutants can be accomplished by one of the several ways, such as adding saboteurs to structural or behavioural component descriptions, mutating structural component descriptions and manually mutating behavioural component descriptions to achieve complex and detailed fault models [59]. Figure 3.3(a) shows a SystemC-based example of original D-type flip-flop and Figure 3.3(b) shows the mutant configuration of the same (using manual mutation).

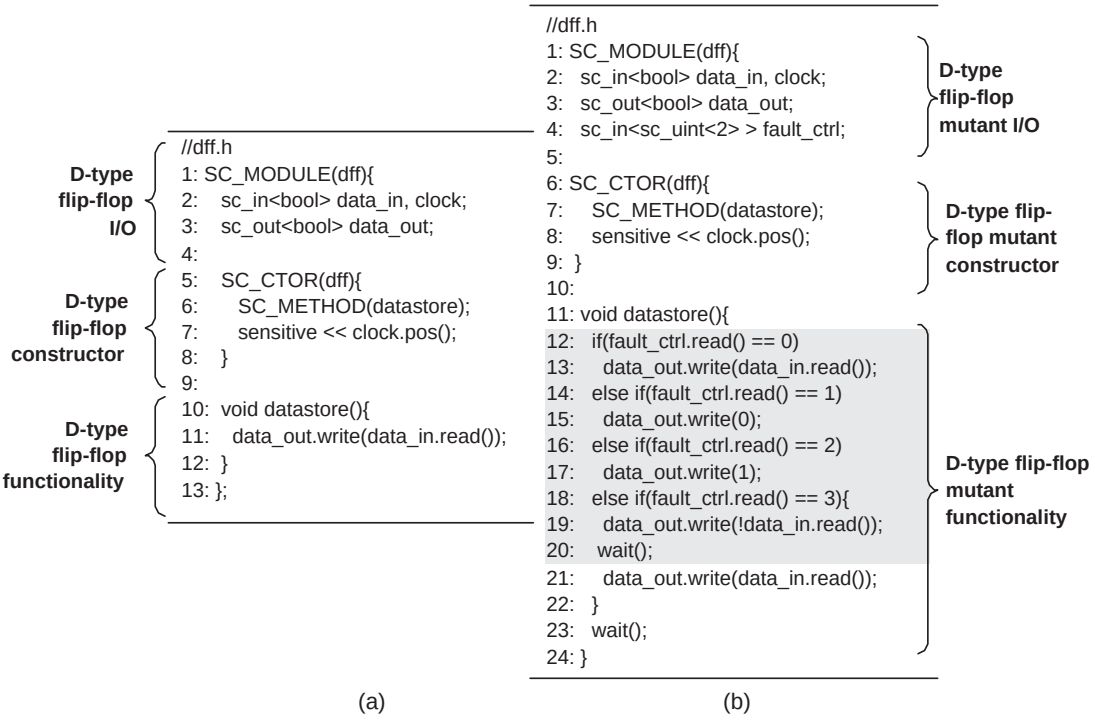


FIGURE 3.3: Example of mutant based fault injection: (a) an original D-type flip-flop in SystemC, (b) a mutant D-type flip-flop in SystemC

As can be seen, the original D-type flip-flop consists of the input/output ports (*data_in*, *clock* and *data_out*), the module constructor (*SC_CTOR(..)*) and the module functionality *datastore(..)* (for brief introduction to module structure in SystemC, see Section 2.4, Chapter 2). The module constructor (*SC_CTOR(..)*) registers the *datastore(..)* functionality as a SystemC *SC_METHOD* process making it sensitive to clock's positive edge (lines 6-7, Figure 3.3(a)). The mutant configuration of D-type flip-flop consists of the basic components within the module (Figure 3.3(b)). To control fault injection the mutant configuration of the D-type flip-flop incorporates *fault_control* signal (line 4). The actual control of fault injection is carried out within the *datastore(..)* functionality of the mutant configuration (lines 11-22). Within *datastore(..)*, when the *fault_control* has a value of 0, no fault is injected and mutant configuration works similarly as the original fault-free configuration (lines 12-13, Figure 3.3(b)). However, when *'fault_control'* has a value of 1, 2 or 3, fault of type stuck-at-0, stuck-at-1 or single-event upset is injected (lines 14-19, Figure 3.3(b)).

Similar to saboteurs, mutants require time consuming re-design or modifications (as shown in the example, lines 12-20, Figure 3.3(b)). Such re-design can become complex as the design description becomes large. To simplify mutation for such designs, SystemC-based auto-mutation tools have been proposed by [115]. In [109, 116] SystemC-based external signal manipulation for simulation of attack-based fault scenarios using mutant-based fault injection models have been shown. In [117] SystemC-based fault injection is carried out using processor and memory mutant models. A number of other illustrative examples of mutant-based approach are presented in [118].

3.1.3 Emulation Techniques

Emulation techniques are based on implementing a close-to-real scenario for fault injection into the device under test (DUT). Two main approaches are used when considering an emulation-based fault injection. The first approach takes advantage of partial run-time reconfiguration available in modern FPGAs, such as [119]. The second approach

Original

```
1: If(data_in > rmax){
2:          rmax = data_in;
3: }
```

(a)

After Emulation

```
1: If(inject_error(inject_error(data_in, error, 8, 11, “1111”, 12, 25, “1111”)) >
inject_error(inject_error(rmax, error, 17, 19, “1111”, 12, 25, “1111”)){
2:          rmax = inject_error(data_in, error, 26, 29, “1111”, 30, 33, “1111”);
3: }
```

(b)

FIGURE 3.4: Example of modification of design description using emulation technique: (a) fault-free (before emulation) design description, and (b) faulty (after emulation) design description

relies on instrumentation, i.e. modifications of the original description to enable fault injection, such as [111]. For all emulation techniques, the fault injection arrangement requires compatibility and necessary translations for use with interface hardware and software tools that are involved in the emulation technique. Emulation-based techniques using the second approach generally restrict the abstraction of system modelling. For example, the design of the DUT needs to be completely synthesizable for the emulation-based technique proposed in [111]. The emulated and automatically re-generated design codes using this technique are changed from the original descriptions. Figure 3.4 shows an example of such emulation technique with the original design description and the modified design description after enabling fault injection through [111]. As can be seen, the modified design description enables fault injection using automatic insertion of

inject_error(..) function to manipulate the variables *data_in* and *rmax* (lines 1-2, Figure 3.4(b)). Comparing between Figure 3.4(a) and (b), it is evident that such technique inflates the design description and has high level of design intrusion. As a result, such emulation technique has limited applicability in validation of system reliability [59].

3.1.4 Simulation Command

Simulation command based approach, which is also employed in the proposed fault simulator, is an effective way to inject faults into signal or variable registers. It requires no or minimum changes in the original design description of the device under test (DUT). The variable manipulation technique allows injection of faults into behavioural models by altering values of variables defined in the design description. In signal manipulation technique, faults are injected by altering the value of signals. Signal manipulation is implemented by disconnecting the signal from its driver and forcing it to a new value for the fault duration. When the fault duration time is over, the signal is connected back to the driver [101]. Varying the fault duration different types of faults can be simulated using such approach, viz. temporary, intermittent and permanent faults, etc.

Recently, SystemC simulation command based approach have been reported by [110, 118], which employ the introduction of extra values within SystemC type *sc_logic*. The extra values are: *SC_LOGIC_A* meaning a stuck-at-1, *SC_LOGIC_B* meaning a stuck-at-0, and *SC_LOGIC_R* meaning *reset* for stuck-at faults. Using these additional values for *sc_logic* along with the original values fault-free and faulty scenarios can be simulated. To enable fault injection in different variable and signal registers within a DUT the original types are required to be changed to the proposed *sc_logic* or its variants (such as *sc_lv*). However, in behavioural models of a DUT different variable and signal register types are common. Hence, restricting them to only *sc_logic* type or its variants limits the applicability of this approach for design descriptions with different variable and signal types. An 8-bit counter example using the simulation command based approach [118] is shown in Section 3.3 and it is then compared with the fault injection technique employed in the proposed fault simulator, which also employs simulation command based approach. The new SystemC fault injection simulator is described next.

3.2 Fault Injection Simulator

The SystemC fault injection simulator implementing a novel fault injection technique has four major components as shown in Figure 3.5. Brief description of each component follows. The actual fault injection technique is described in Section 3.2.3.

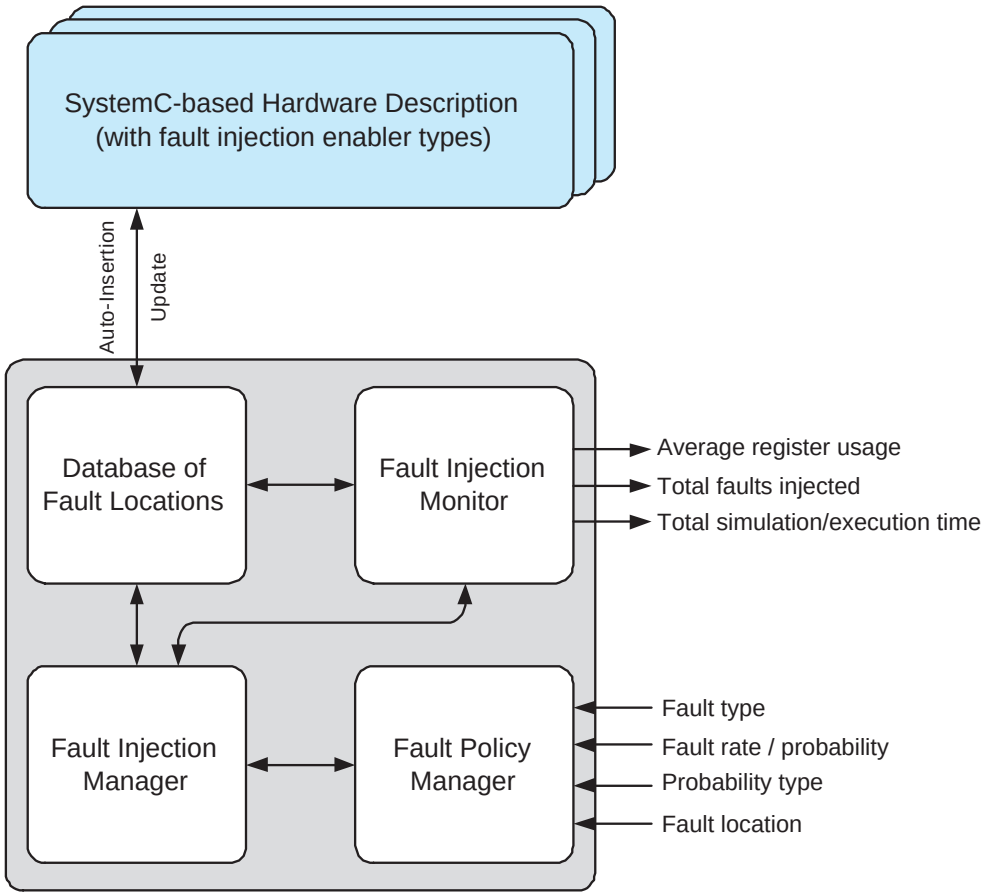


FIGURE 3.5: Block diagram of the proposed SystemC fault injection simulator

3.2.1 Fault Locations Database

Fault locations database is the first module within the proposed fault simulator. It contains the organised list of locations of the registers, where faults can be injected. The database is formed by automated insertion and update of fault locations (variable or signal registers) from the original SystemC design description (Figure 3.5). The automated insertion in fault locations database is initiated by replacing the original variable and signal register types to the fault injection enabler types. Table 3.1 shows the proposed fault injection enabler types for commonly used primitive and SystemC types. As can be seen, the original primitive types, for example, *int* and *long* are replaced by *Reg*<*int*> and *Reg*<*long*>. Also, the SystemC types, for example, *sc_logic* and *sc_int*<*N*> are replaced by fault injection enabler types *LogicReg* and *IntReg*<*N*> (Table 3.1). The fault injection enabler types in Table 3.1 employ careful and transparent implementation of their original types, such that their functions are kept intact. To enable automated insertion of the registers, the fault injection enabler types use constructor (or initialisers). During initialisation in the constructor, the register locations of the variable or signal types are inserted in the fault locations database. When the scopes of these replacement types are expired, destructors (or de-initialisers) are used, which remove the the register

Original types	**Fault injection enabler types**
primitive type (int, bool etc.)	Reg<primitive type>
sc_bit	BitReg
sc_logic	LogicReg
sc_fix	FixReg
sc_ufix	UFixReg
sc_int<N>	IntReg<N>
sc_uint<N>	UIntReg< *N* >
sc_bigint<N>	BigIntReg<N>
sc_biguint<N>	BigUIntReg<N>
sc_lv<N>	LogicVectorReg<N>
sc_bv<N>	BitVectorReg<N>
sc_fixed<N>	FixedReg<N>
sc_ufixed<N>	UFixedReg<N>

TABLE 3.1: Original variable/signal types and corresponding fault injection enabler types used in the fault injection simulator

locations from the fault locations database (Figure 3.5). Since database updates are possible from multiple design modules at the same time, the database implementation is made thread-safe by singleton design pattern within the simulator. Such design pattern restricts mutually inclusive database updates at the same time. Using such thread-safe implementation, the fault locations database or the register space within the proposed fault simulator, *FIMgr*, can be accessed through the following SystemC statement:

```
>> database = FIMgr::getInstance().register_space;
```

To demonstrate how the fault injection enabler types are implemented, Figure 3.6 shows the SystemC definitions of proposed *Reg<primitive type>* and *IntReg<N>* types, which are equivalent to primitive types and SystemC type *sc_int<N>*. As can be seen, the constructor (*Reg(..)*) in the SystemC definition of *Reg<primitive type>* inserts the register location of the original type in a centralised fault locations database within the fault simulator (called *FIMgr*) (lines 6-12, Figure 3.6(a)). This is done by calling the member function *registerInsert(..)* of *FIMgr* with appropriate size and data type information (line 9-10). The constructor is called upon every time a variable or signal register is initialised. When the signal or variable register is de-initialised, destructor (*~Reg(..)*) is called upon. Within the destructor, the register location is removed from the fault locations database, using *registerDelete(..)* member function of *FIMgr* (lines 14-17, Figure 3.6(a)). The original functionality of primitive types are implemented in *Reg<primitive type>* through a set of SystemC operator definitions (line 18, Figure 3.6(a)). Due to usage of SystemC template, the *Reg<primitive type>* definition

```
//Reg.h
1:  #define BYTE 8 //byte->bits
2:  #define size(x) ((int) (sizeof(x) * BYTE))
3:  template <typename T>
4:  struct Reg{
5:    T reg;
6:    //Constructor
7:    Reg(T _reg = 0) {
8:      reg = _reg;
9:      FIMgr::getInstance().registerInsert
10:     ((void *) &reg, size(reg), 11: DATA_TYPE);
11:  }
12:
13:  //Destructor
14:  ~Reg() {
15:    FIMgr::getInstance().registerDelete
16:    ((void *) &reg);
17:  }
18:  // Inline operator implementation functions
19: };
```

(a)

```
//IntReg.h
1:  template <int W>
2:  class RegInt : public sc_int_base{
3:    ....
4:    //Constructors..
5:    IntReg():sc_int_base(W){
6:      FIMgr::getInstance().registerInsert
7:        ((void *) &m_val, W, DATA_TYPE);
8:    }
9:    IntReg(int_type v):sc_int_base(v, W){
10:     FIMgr::getInstance().registerInsert
11:       ((void *) &m_val, W, DATA_TYPE);
12:   }
13:     ......
14:   //Destructor
15:   ~IntReg(){
16:     FIMgr::getInstance().registerDelete
17:       ((void *) &m_val);
18:     }
19:   //Other original functions
20: };
```

(b)

FIGURE 3.6: SystemC definition of (a) *Reg<primitive type>* as a replacement of primitive types, (b) *IntReg<N>* as replacement of SystemC *sc_int<N>* type for enabling fault injection

can accept any primitive type as a parameter of *Reg<primitive type>*, such as *int*, *long* (line 3, Figure 3.6(a)). Similar to *Reg<pritimitive type>*, the overloaded constructors of *IntReg<N>* inserts register location of the value-holder variable *m_val* into the fault locations database by calling the *registerInsert(..)* function with corresponding size of data in bits and their types (lines 4-12, Figure 3.6(b)). During de-initialisation the destructor is called upon, which uses *registerDelete(..)* to remove the register location from the fault locations database (lines 14-18, Figure 3.6(b)). The original functionality of *IntReg<N>* is implemented in a transparent way by overloading their operator-related functions (line 19, Figure 3.6(b)).

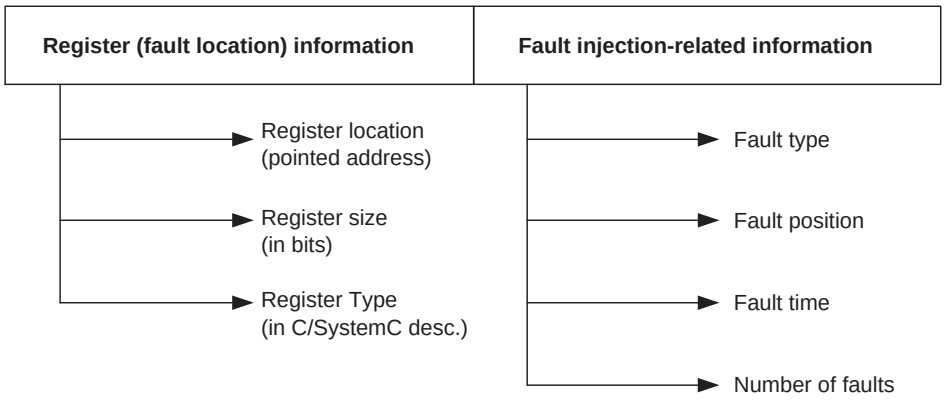


FIGURE 3.7: Organisation of each record within the fault locations database

Upon insertion of variable and signal register locations into the fault locations database, they are organised with related information to facilitate the fault injection. Figure 3.7 shows how each record within the fault locations database is organised. The information related to each register is organised in two major categories: information related to the register (the possible fault location) and information related to fault injection. The register information contains the location of the register, it's size and the original type of the register (Figure 3.7). For example, through the use of *Reg<long> x;* instead of *long x;* in the SystemC design description, the fault injection technique employed in the proposed simulator inserts into the fault locations database the address of x, the size as 64 bits and data type as 'LONG' (a pre-defined constant indicating the original type of *long*). Each entry in the database also contains information to enable fault injection, such as the fault type, the position of the fault, the fault injection time and the number of faults injected (Figure 3.7). The organisation within the fault locations database provides an effective and fast technique for search within the database to facilitate fault injection and fault policy management, as explained in the following sections.

3.2.2 Fault Policy Manager

The fault policy manager (FPM) is the second module of the proposed fault simulator (Figure 3.5). The FPM interacts with the fault injection manager (FIM) and provides flexibility of simulating different fault scenarios. The different fault injection policies are managed within the FPM using the following specifications from the fault injection manager:

- *Fault type*: This specifies the type of fault model to be simulated. Currently, the simulator supports simulation of stuck-at (permanent) faults, bit flips or SEUs, indetermination and delay type (transient) faults, which are reckoned as the major fault types used in fault injection campaigns (for introduction to different fault types, see Section 2.3, Chapter 2). The actual fault injection mechanism for these fault types are described in Section 3.2.3.

- *Fault probability or rate*: This defines the rate at which the fault injection takes place. Fault rate within the FPM is specified by either fault probability (i.e. faults per bit) or fault injection rate (i.e. faults per bit per clock cycle). For transient faults, such as SEUs, this specifies the timing and location of fault. For stuck-at faults, this rate defines the number of faults per bit (for introduction to different fault injection rates, see Section 2.3, Chapter 2). Determination of fault rate for different types of faults is detailed in Section 3.2.3

- *Fault probability type*: The FPM facilitates probabilistic fault simulation using fault rate, which is often used in fault injection experiments [60, 61]. A probability distribution is also specified within the FPM module, which determines the

location of faults within the fault locations database. Poisson's distribution is generally used for determining fault location [62, 91] as the register space (collection of possible fault locations) is formed of dynamically updated application registers. Also, exponential distribution is generally used for determining the timing of fault injection using single-event upset model [64].

- *Fault Location*: This specifies the desired fault location within the available registers. Such specification is useful for manual fault injection in a target register within a DUT.

With the above specifications, the FPM module manages different fault injection scenarios and interfaces with fault injection manager for fault injection. Figure 3.8 shows the SystemC implementation of the FPM module, *FIPolicy*, with the main functions. The FPM member functions *setFaultType(..)* (line 6), *setFaultProbability(..)* (line 7) and *setFaultProbabilityType(..)* (line 8) take input from the top-level SystemC module or testbench for the fault type (*fault_type*), probability (*prob*), and its type (*probType*) for the fault injection. The other member functions *getSearchLength(..)* (lines 10-13) and *getFaultLocation(..)* (lines 15-22) return to the fault injection manager (FIM) module the associated search length, *search_length*, and fault location within the available bit space in the fault locations database based on the fault probability (p_B). The generation of random fault location is implemented using the following random probability distribution functions: exponential, uniform, and Poisson's. The different probability distributions are implemented using GNU scientific library [120] (*gsl_rng*, line 3, Figure 3.8).

Using the FPM module, the proposed fault simulator supports two fault injection modes: manual fault injection and probabilistic fault injection. Manual fault injection requires the user to specify fault injection within the design description of the DUT. To carry out manual fault injection in a target signal and variable register (*reg*), the location of the register, *fault_loc*, is obtained first by

```
>> long fault_loc = FIMgr::getInstance().getLocation(DUT.reg.get_ptr());
```

Later, with the given location, *fault_loc*, the fault location is specified through the FPM within the fault simulator by

```
>> FIMgr::getInstance().getPolicy().setFaultLocation(fault_loc);.
```

Finally, the manual fault injection of the given type, FAULT_TYPE, is carried out by

```
>> FIMgr::getInstance().inject_faults(FAULT_TYPE);.
```

```
//FIPolicy.h
1:  class FIPolicy{
2:    const gsl_rng_type * T;
3:    gsl_rng * r;
4:    long search_length;
5:    ……..
6:    void setFaultType(int _fault_type){fault_type = _fault_type;}
7:    void setFaultProbabilityType(int _probType){probType = _probType;}
8:    void setFaultProbability(double _prob){prob = _prob;}
9:
10:  long getSearchLength(){
11:    search_length = (prob > 0) ? (1/prob) : -1;
12:    return search_length;
13:  }
14:
15:  long getFaultLocation(){
16:    switch(probType){
17:      case UNIFORM:
18:        fault_loc = (long) (gsl_rng_uniform(r) * search_length); return fault_loc;
19:      case POISSON:
20:        fault_loc = (long) gsl_rng_poisson(r, (double)(search_length / 2));
21:        return fault_loc;
22:    }
23:  }
24:  ……..
25:  void setFaultLocation(long _fault_loc){fault_loc = _fault_loc;}
26:   ….
27:};
```

FIGURE 3.8: SystemC model of the fault policy manager, *FIPolicy* showing main functions

Manual fault injection is useful to validate the system reliability with faults in the target registers. However, for validation of reliability of large or complex systems, probabilistic fault injection is a popular technique [61]. The probabilistic fault injection requires specification of the fault type, fault rate and specified probability before the actual fault injection takes place. An example illustration of how the FPM functionality (shown in Figure 3.8) is used within the top-level SystemC module or testbench to initiate probabilistic fault injection follows. For example, to specify the type of fault as single-event upset in the current policy, the following SystemC statement is issued within the top-level module or testbench

```
>> FIMgr::getInstance().getPolicy().setFaultType(SEU);.
```

Next, to inject faults at the rate of FAULT_RATE with Poisson's distribution within the fault locations database, the following SystemC statements are used within the the top-level module or testbench

```
>> FIMgr::getInstance().getPolicy().setFaultProbability(FAULT_RATE);

>> FIMgr::getInstance().getPolicy().setFaultProbabilityType(POISSON);
```

With the above specifications, the probabilistic fault injection starts with the SystemC simulation start command *sc_start(..)* in the top-level module. The actual fault injection is managed by the fault injection manager (FIM) module and is described next.

3.2.3 Fault Injection Manager

The fault injection manager (FIM) is the third and main module of the proposed fault simulator (Figure 3.5). It interfaces with the FPM module and the fault locations database to inject faults implementing a novel fault injection technique. The FIM module interfaces with the FPM module for information related to the fault type, rate and associated probability (Section 3.2.2). Also, it interfaces with the fault locations database, which creates a centralised register space for fault injection (Section 3.2.1). The fault location within the register space and the fault injection timing are determined from the specified fault probability or rate, p_B. The number of bits searched in each clock cycle is equal to the total number of registers within the fault locations database. Figure 3.9 shows a diagrammatic demonstration of how the register space and available simulation time are sampled for the fault injection. As can be seen, in each clock period

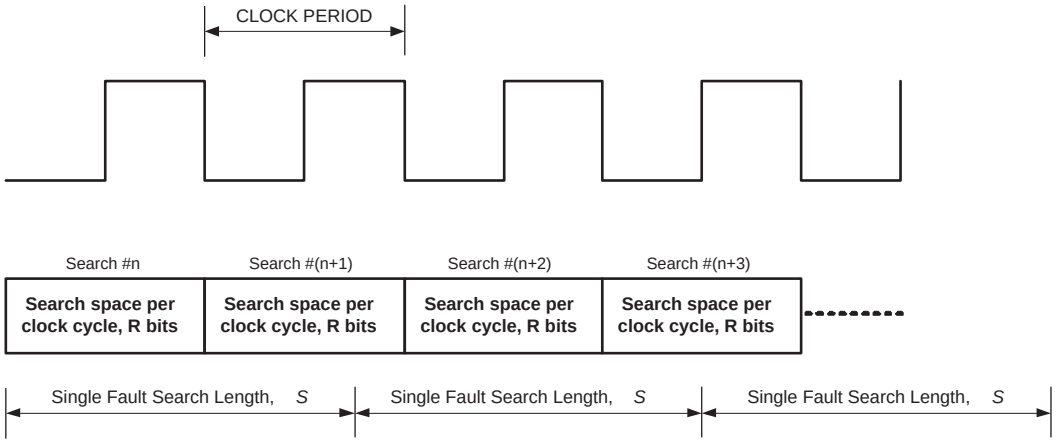


FIGURE 3.9: Fault injection mechanism using sampling between time and register space

a complete search within the register space, R (in bits), is performed (Figure 3.9). The search continues iteratively until a single fault is injected within the specified search length, S (in bits) (Figure 3.9). To incorporate the timing information in the fault injection simulator, the FIM module requires access to the system clock. This is done by connecting the the system clock with the FIM using the following SystemC statement:

```
>> FIMgr::getInstance().clk(top_level_clock);
```

Figure 3.10 shows SystemC description of the fault injection technique within the FIM module, *FIManager*. The fault injection is implemented using the fault injection func-

tion, *inject_faults*, which employs the time sampled search within register space (Figure 3.9). As shown in Figure 3.10, using the member functions *getFaultLength(..)* and

```
#include “FIReg.h” #include “ FIPolicy.h”
1:  class FIManager: public sc_module{
2:    .....
3:    SC_CTOR(FIManager){
4:      .....
5:      SC_CTHREAD(inject_faults, clk.pos();
6:    }
7:    void inject_faults(){
8:      .....
9:      list<RegisterElement>::iterator listIterator;
10:    long loc_counter;
11:    while(true){
12:      wait();
13:      search_length = currentPolicy.getSearchLength();
14:      fault_loc = currentPolicy.getFaultLocation();
15:      loc_counter = 0;
16:      for(listIterator = register_space.begin(); listIterator != register_space.end();){
17:        if(fault_loc >= listIterator.loc_counter && fault_loc <= listIterator.search_length)
             *(TYPE*)(*listIterator).registers=*(TYPE*)(*listIterator).registers^(TYPE)fault;
18:        loc_counter+=(*listIterator).size_bits; listIterator++;
19:        if(loc_counter < search_length && lisIterator == register_space.end())
20:          {wait(); listIterator = register_space.begin();}
21:      }……..
22:    }
23: }
24:  .....
25:  FIPolicy currentPolicy; list<RegisterElement> register_space;
26:};
27:typedef Singleton<FIManager> FIMgr;
```

Include database and policy manager class

FIM constructor

FIM fault injection

FIM Fault locations database, *register_space*

FIGURE 3.10: SystemC model of the fault injection manager showing main functions

getFaultLocation(..) of the fault policy manager (FPM) module, the search length within the register space, *search_length* (of S bits) and fault location, *fault_loc*, is found (lines 13-14). For single fault injection or clearance, the fault locations database (*register_space*) is searched in each iteration (lines 16-21). The iteration continues until the location counter, *loc_counter*, reaches the search length, (*search_length*) (line 19-20). To speed up iterative searching process, the iteration in which the fault injection would not be done are skipped with appropriate increase of the *loc_counter*. Once the *fault_loc* is found in the current iteration, the fault is either injected or cleared depending on the type of fault (line 17). During the fault injection the registers in the *register_space* are converted back to their original types within the FIM (line 17). Since the fault injection technique implemented on a prototype simulator requires search through the *register_space* on every clock cycle, the member function of the FIM module *inject_fault* is registered as a SystemC process sensitive to the clock (line 5). To provide single instance of the FIM across all other design components within the DUT, a singleton instance of the FIM module, *FIMgr*, is generated (line 27, Figure 3.10). The top-level module, testbench and the SystemC design description of the DUT uses this *FIMgr* as a global single instance. Multiple instances can be incorporated by creating number of such singleton

objects.

Table 3.2 shows how faults are injected for various fault types with relevant parameters. For simplicity, symbol for each parameter is used: S meaning *search_length*, r meaning *fault_loc*, τ meaning *fault_duration* and r' meaning *approx_fault_loc*. As can be seen from

Fault types	**Fault injection technique**
Stuck-at-fault	A bit at r in S is changed to '1' or '0' for stuck-at-1 or stuck-at-0 faults for τ cycles (if temporary) or indefinitely (if permanent)
Bit-flip/SEU	A bit at r in S is XORed with '1' for one clock cycle duration (i.e. $\tau = 1$)
Delay	Original SystemC *wait()* statements are replaced by $wait(N)$ statements, where N is registered in the fault locations database. A bit close to r' in S is bit flipped for such types for specified delay of τ clock cycles
Indetermination	A logic type value close to r' in S is changed from '0'/'1' to 'X' for τ cycles (if temporary) or indefinitely (if permanent)

TABLE 3.2: Different fault types and injection techniques used in the fault injection manager (FIM)

Table 3.2, FIM within the proposed fault simulator can flexibly implement different fault types using register perturbation within the fault locations database. The transient faults are injected by altering the original r-th or r'-th bit and cleared after a duration of τ cycles using XOR operation with '1'. For permanent stuck-at faults, the alterations are not cleared throughout the simulation time.

Since the fault injector performs one search per clock cycle in the *register_space* (Figure 3.9), the fault probability, p_B (faults per bit) is interpreted as the injection of one fault in *search_length* of $S = \left\lceil \frac{1}{p_B} \right\rceil$ bits or two concurrent faults in every $S = \left\lceil 2 \times \frac{1}{p_B} \right\rceil$ bits and so on. Hence, the injection of one fault requires the search to be carried out over $\left\lceil \frac{1}{p_B R} \right\rceil$ clock cycles, where R is the average register usage or average size of the fault locations database (in bits per clock cycle). Assuming the fault duration (for temporary faults) of τ clock cycles only, the average time required for one transient fault injection, $T_{transient}$, is given by

$$T_{transient} = \left\lceil \frac{1}{p_B R} \right\rceil + \tau \quad . \tag{3.1}$$

The register usage of the device under test (DUT), R, is defined as

$$R = \frac{1}{T_E} \sum_{t=1}^{T_E} R_t \quad , \tag{3.2}$$

where R_t is the size of the register space at t-th clock cycle (found by logging the different sizes of register space over simulation time in the fault injection monitor, Section 3.2.4). Employing the average fault injection time from (3.1), the total number of transient faults injected, $\Gamma_{transient}$, in total execution time of T_E is given as

$$\Gamma_{transient} = \left\lceil \frac{T_E}{T_{transient}} \right\rceil \approx \left\lceil \frac{T_E}{\left\lceil \frac{1}{p_B R} \right\rceil + \tau} \right\rceil . \tag{3.3}$$

Dividing the total faults injected in (3.3) by the execution time (T_E) and average register usage (R), the transient error rate or fault injection rate per clock cycle per bit, $\lambda_{transient}$, is given as

$$\lambda_{transient} = \left\lceil \frac{1}{R \times T_{transient}} \right\rceil \approx \left\lceil \frac{1}{R \times \left(\left\lceil \frac{1}{p_B R} \right\rceil + \tau \right)} \right\rceil . \tag{3.4}$$

Equation (3.4) gives the soft error rate (SER) that is often used as a unit for fault injection experiments using soft error models [62, 121] (also used in Chapters 4, 5 and 6). Depending on the duration of the transient fault, τ, $\lambda_{transient}$ in (3.4) can vary for different fault types. For example, assuming a single-event upset (SEU) model for transient fault (implying $\tau = 1$), the soft error rate (SER) is given by

$$\lambda_{SEU} = \left\lceil \frac{1}{R \times T_{SEU}} \right\rceil \approx \left\lceil \frac{1}{R \times \left(\left\lceil \frac{1}{p_B R} \right\rceil + 1 \right)} \right\rceil . \tag{3.5}$$

Using (3.4) the transient fault probability, p_B (fault per bit), can be defined in terms of transient fault injection rate, $\lambda_{transient}$ (fault per bit per clock cycle), as

$$p_B \approx \frac{1}{\left(\frac{1}{\lambda_{transient}} - \tau R \right)} \quad , \tag{3.6}$$

assuming that $\frac{1}{(p_B R)} >> 1$ or $(p_B R) << 1$. For permanent fault models, such as stuck-at faults, the fault rate is expressed in terms of fault probability, p_B. The total number of permanent faults injected by the proposed fault injection simulator is given as

$$\Gamma_{permanent} = p_B R \quad . \tag{3.7}$$

Equation (3.3) and (3.7) give total faults injected for transient and permanent faults. To demonstrate how many faults would be injected for a given fault injection rate using the fault injection technique employed in the proposed fault injection simulator, two examples follow.

Example 1: Given a system with average size of register space or register usage of

R=10k bits and execution time of T_E=1×10^6 clock cycles, the total number of faults injected in the given system assuming single-event upset (SEU) model (i.e. $\tau = 1$ clock cycle) with fault probability of p_B=1×10^{-8} (faults per bit) is found by (3.3) as Γ=100 SEUs. Using (3.5) the equivalent SER of p_B=1×10^{-8} is given as λ=1×10^{-10}.

Example 2: Given a system with average size of register space or register usage of R=10k bits and execution time of T_E=1×10^6 clock cycles), the total number of faults injected in the given system assuming stuck-at fault model with fault probability of p_B=1×10^{-4} (faults per bit) is found by (3.7) as 1 stuck-at fault.

3.2.4 Fault Injection Monitor

The fault injection monitor is the final component of the proposed fault simulator (Figure 3.5). It interfaces with the fault locations database and the FIM module to provide with the following useful simulation specific outputs:

- *Total simulation or execution time*: Since the clock of the DUT is connected to the FIM within the proposed fault simulator (Section 3.2.3), the fault injection monitor can output the observation time for fault injection as simulation or execution time, denoted by T_E.
- *Average register usage*: The average register usage, R, gives a measure of average number of registers that are subjected to fault injection per clock cycle. This is the found through logging and averaging the length of the register space in the fault locations database over the execution time, T_E, and is given by (3.2).
- *Number of total faults injected*: For a given fault injection rate or probability (p_B), specified through the FPM module (Section 3.2.2), the total number of faults injected, Γ, is recorded by the fault injection monitor. An approximate analytical measure of the total number of transient faults injected for a given fault rate is given by (3.3) and total number of permanent faults injected for a given fault probability is given by (3.7).

Enabling the fault injection through the proposed types (Table 3.1) and managing the fault injection and policies within the fault injection manager, a prototype fault simulator is developed. In the following, the effectiveness of the fault injection technique implemented on the a prototype fault simulator is described and it is compared with the existing SystemC-based fault injection techniques.

3.3 Comparison of Fault Injection Techniques

The effectiveness and capabilities of the technique employed in the proposed fault injection simulator is demonstrated through comparison against other techniques: saboteur [60], mutant [118] and simulation command based approach [110]. Using SystemC models of synchronous 8-bit counter and a D-type flip flop, three comparisons are carried out in terms of the following: 1) simplicity and intrusiveness, 2) fault representation and capabilities, and 3) simulation speed. The detailed comparisons follow.

3.3.1 Comparison 1: Simplicity and Intrusiveness

As described in Section 3.1, a highly intrusive re-design for enabling fault injection into the DUT is not desirable. The technique in the proposed simulator can be applied in the design description of a DUT with minimum intrusion (i.e. minimum design modification). To demonstrate this, an original synchronous 8-bit counter design in SystemC and the modified design using the technique employed in the proposed fault simulator are presented in Figure 3.11. As can be seen, the technique employed in the proposed simulator requires simple replacement of the original types *bool* and *sc_uint<8>* to their equivalent fault injection enabler types (Table 3.1) *Reg<bool>* and *UIntReg<8>* (compare between lines 3-6, Figure 3.11(a), and lines 4-7, Figure 3.11(b)). The inclusion of the SystemC header file *FIReg.h* (line 1, Figure 3.11(b)) enables the usage of the fault injection enabler types in the 8-bit counter description. The new signal or variable types allow automatic update of the fault locations database (Section 3.2.1) and fault injection within the FIM (Section 3.2.3), keeping the rest of the design intact without changing its functionality.

To compare the technique employed in the proposed fault injection simulator with other techniques in terms of simplicity and intrusiveness, SystemC design of synchronous 8-bit counter of saboteur [60], mutant [118] and simulation command based approach [110] are shown in Figure 3.12. Each of these descriptions give similar fault injection capabilities as the fault injection technique implemented on a prototype fault injection simulator. Comparing SystemC descriptions in Figure 3.11(a) with Figure 3.12(a)-(b), it can be seen that saboteurs and mutants have the highest level of design intrusion. The saboteur-based fault injection is implemented with the insertion of three serial simple saboteurs: *sab_reset*, *sab_enable* and *sab_counter* (Figure 3.12(a)). Using these saboteurs the fault injection is controlled for *reset*, *enable* and *counter* signals. However, these saboteurs cannot inject faults into variable register *count*, since it is not a signal (Section 3.1.1). The mutant-based fault injection is implemented with the insertion of the same saboteurs with intrusive design description to inject faults into the variable register *count* (Figure 3.12(b)). For example, the behavioural faults are injected at local register *count* by using the function, *inject_bfaults* (lines 18-30, Figure 3.12(b)). Different

```
1:  SC_MODULE (counter8bit) {
2:    sc_in_clk clock ;
3:    sc_in<bool> reset ;
4:    sc_in<bool > enable;
5:    sc_out<sc_uint<8> > counter_out;
6:    sc_uint<8> count;
7:
8:    void incr_count() {
9:      while(true){
10:       if (reset.read() == 1) {
11:         count = 0; counter_out.write(count);
12:       }
13:       else if (enable.read() == 1) {
14:         count = count + 1;
15:         counter_out.write(count);
16:       }
17:       wait();
18:     }
19:  }
20:
21:  SC_CTOR(counter8bit) {
22:    SC_THREAD(incr_count);
23:    sensitive << reset;
24:    sensitive << clock.pos();
25:  }
26:};
```

(a)

```
1:  #include “fim/FIReg.h”
2:  SC_MODULE (counter8bit) {
3:    sc_in_clk clock ;
4:    sc_in<Reg<bool> > reset ;
5:    sc_in<Reg<bool> > enable;
6:    sc_out<UIntReg<8> > counter_out;
7:    UIntReg<8> count;
8:
9:    void incr_count() {
10:     while(true){
11:       if (reset.read() == 1) {
12:         count = 0; counter_out.write(count);
13:       }
14:       else if (enable.read() == 1) {
15:         count = count + 1;
16:         counter_out.write(count);
17:       }
18:       wait();
19:     }
20:  }
21:
22:  SC_CTOR(counter8bit) {
23:    SC_THREAD(incr_count);
24:    sensitive << reset;
25:    sensitive << clock.pos();
26:  }
27:};
```

(b)

FIGURE 3.11: Example illustration of the fault injection technique employed in the proposed fault injection simulator: (a) a synchronous SystemC 8-bit counter module, and (b) a synchronous SystemC 8-bit counter module using the technique employed in proposed fault injection simulator

fault types within the mutant-based counter are controlled externally from testbench by using signal array of *fault_type* (line 6, Figure3.12(b)). The counter design specification using the simulation command-based approach [110, 118] replaces the original *bool* type to *sc_logic* type (lines 3-5) and *sc_uint* type to *sc_lv*<*N*> type (vector of *sc_logic*, line 6, Figure 3.12(c)). These changes of the variable or signal types have implication on the design specification, although the level of design intrusion is kept low. For example, to keep the functionality of the original specification, additional programming effort is required as shown in line 14 (Figure 3.12(c)).

```
1:  #define MAX 4
2:  SC_MODULE (counter8bit) {
3:    //original ports and signals/variables
4:    Saboteur<bool> sab_reset, sab_enable;
5:    Saboteur<sc_uint<8> > sab_counter;
6:    sc_signal<sc_uint<2> > fault_type[MAX];
7:    sc_signal<bool> reset_out, enable_out;
8:    sc_signal<sc_uint<8> > counter_in;
9:    int fault_loc;
10:   sc_uint<8> count;
11:
12:   void incr_count () {
13:     .....
14:     count = inject_bfaults(count.to_int(), 8);
15:     count = count + 1;
16:     counter_out.write(count);
17:   }
18:   int inject_bfaults (int _var, int _size){
19:     int fault_at, fault;
20:     if(fault_type[MAX].read() == NO_FAULT){
21:       return _var;
22:     }
23:     else if(fault_type[MAX].read() == SEU){
24:       fault_at = srand() % _size;
25:       fault = (1 << fault_at);
26:       _var = _var ^ fault;
27:       return _var;
28:     }
29:     ......
30:   }
31:
32:   SC_CTOR(counter8bit): sab_reset("S_R"),
33:     sab_enable("S_E"),sab_counter("S_C"){
34:     sab_reset.sab_in(reset);
35:     sab_reset.sab_out(reset_out);
36:     sab_enable.sab_in(enable);
37:     sab_enable.sab_out(enable_out);
38:     sab_counter.sab_in(counter_out);
39:     sab_counter.sab_out(counter_in);
40:     sab_reset.fault_type(fault_type[0]);
41:     sab_enable.fault_type(fault_type[1]);
42:     sab_counter.fault_type(fault_type[2]);
43:     for(int i = 0; i < MAX; i++){
44:       fault_type[i].write(NO_FAULT);
45:     }
46:     .....
47:   }
48:};
```

(b)

```
1:  SC_MODULE (counter8bit) {
2:    //original ports and signals/variables
3:    Saboteur<bool> sab_reset, sab_enable;
4:    Saboteur<sc_uint<8> > sab_counter;
5:    sc_signal<bool> reset_out, enable_out;
6:    sc_signal<sc_uint<8> > counter_in;
7:    sc_uint<8>  count;
8:
9:    void incr_count(){...}
10:   SC_CTOR(counter8bit): sab_reset("Sab_R"),
11:   sab_enable("Sab_E"), sab_counter("Sab_C") {
12:     sab_reset.sab_in(reset);
13:     sab_reset.sab_out(reset_out);
14:     sab_enable.sab_in(enable);
15:     sab_enable.sab_out(enable_out);
16:     sab_counter.sab_in(counter_out);
17:     sab_counter.sab_out(counter_in);
18:     ....
19:   }
20:};
```

(a)

```
1:  SC_MODULE (counter8bit) {
2:    sc_in_clk clock ;
3:    sc_in<sc_logic> reset ;
4:    sc_in<sc_logic > enable;
5:    sc_out<sc_logic > counter_out;
6:    sc_lv<8> count;
7:
8:    void incr_count() {
9:      while(true){
10:       if (reset.read() == 1) {
11:         count = 0; counter_out.write(count);
12:       }
13:       else if (enable.read() == 1) {
14:         count = count.to_int() + 1;
15:         counter_out.write(count);
16:       }
17:       wait();
18:     }
19:   }
20:
21:   SC_CTOR(counter8bit) {
22:     SC_THREAD(incr_count);
23:     sensitive << reset;
24:     sensitive << clock.pos();
25:   }
26:};
```

(c)

FIGURE 3.12: Example of fault injection in synchronous 8-bit counter design using (a) saboteurs, (b) mutant, and (c) simulation command based approach

Apart from additional programming effort, the type changes shown in Figure 3.12(c) have a number disadvantages compared to the proposed fault injection technique (Figure 3.11(b)). These are as follows.

1. Instead of only two ('0' and '1') values for *bool*, each replaced type *sc_logic* will now have seven values, $'0'$, $'1'$, $'X'$, $'Z'$, $'A'$ (stuck-at '1'), $'B'$ (stuck-at '0') and $'R'$ (stuck-at reset), with default value of $'X'$. This default value ('X') gives undefined initial state of the signal or variable during initial cycles on the SystemC simulation. To avoid undefined value in early simulation cycles in SystemC, *sc_start(..)* statement to start simulation needs to be changed to *sc_initialize(..)*. The following simulation cycles are then controlled by *sc_cycle(..)* statement. In the proposed fault injection simulator, value space of the changed type *Reg<bool>* remains unchanged, while fault injection is enabled by replacing the original types to fault injection enabler types (Table 3.1) and the original simulation control statements require no change, viz. the simulation can be initialised by default *sc_start(..)* SystemC statement.

2. Since the original type and the re-defined *sc_logic* or *sc_lv<N>* type is not directly compatible with each other, the designer needs to make manual code modifications. The required modifications for achieving equivalence between the original and the changed types inflate the design description. An example of such modification is shown line 14 (Figure 3.12(c)). Often limiting original types to *sc_logic* or *sc_lv<N>* type is not feasible for some types, for example integer array type (*int[..]*), or floating point type (*float*) or it's array type (*float[..]*) etc. The technique employed in the proposed simulator has transparent and equivalent implementations for all variable and signal register types, requiring no modification in how they are used in different scopes within the design description of a DUT.

From the above examples, it is evident that the fault injection technique employed in the proposed fault simulator is less intrusive compared to other reported fault injection techniques: saboteur [60], mutant [110] or simulation command based approach [118]. The minimum intrusive modification of the design description makes the proposed fault injection simulator simple but effective.

3.3.2 Comparison 2: Fault Representation and Capabilities

Faults injection using simulation command based approach is implemented mainly by perturbation of registers [59]. In behavioural or higher-level description of systems, these faults are propagated from variable or data registers to signal registers or vice versa, often referred to as multiplicity [57]. As such a desirable feature of a fault injection technique is to have access to all variable and signal registers of a model description for better

fault access and representation. Saboteurs, as shown in Section 3.1.1, have no access to variable manipulation and therefore, have poor fault representation for behavioural systems, An example of this is shown in Figure 3.12(a), where the variable register *count* is not accessible by any fault. Mutants require extra manual or behavioural modelling to introduce variable manipulation (Figure 3.12(b)). Simulation command based approach [110] restricts types to *sc_logic* and *sc_lv<N>* for fault injection. As a result, it cannot be used to replace all variable register types (Section 3.3.1). The technique employed in the proposed simulator can be applied to replace any primitive or SystemC type to fault injection enabler types (Table 3.1) giving it full access to all variable and signal registers within a DUT.

To compare the proposed fault simulation technique with the other techniques in terms of different capabilities, the following features of a fault simulator are considered:

- *Design abstraction*: System-level modelling is carried out at different abstraction-levels. Hence, a desirable feature of fault simulators is to be able to enable fault injection for systems at various design abstractions [122]. However, some fault simulators restrict the abstraction level of the DUT design description to enable fault simulations. For example, VHDL-based fault injection technique using automated codes insertion proposed in [101] and SystemC-based fault injection technique using simulation command based approach proposed in [118] require that the original description should be in register transfer-level (RTL). Similarly, emulation-based fault injection technique proposed by [111] require that the original description should be completely synthesizable. The proposed fault simulator works by replacing the original variable or signal types to fault injection enabler types and does not impose any such restriction to the design abstraction.

- *Fault types*: The proposed fault injection simulator can simulate the major fault types, including SEUs, stuck-up faults, indetermination and delay faults (described in Section 3.2.3), which is considered as a desirable feature of a fault simulator [57].

- *Fault injection mode*: Another useful feature of a fault simulator is to be able to carry out fault injection in different modes: manual or probabilistic fault simulations [57]. Manual fault simulations are useful for fault injection to target registers of a system to evaluate the reliability in the presence of different types of faults [59]. However, for validation of reliability in the presence of transient faults, particularly due to SEUs, probabilistic fault simulation has been reported [60, 123]. The proposed fault injection simulator can carry out both manual and probabilistic fault simulations (Section 3.2.2). Similar fault injection capability using SystemC is reported in [60]. The fault injection technique using simulation command based approach [110] employs manual fault injection.

3.3.3 Comparison 3: Simulation Time

Simulation time is often used as a benchmark for simulator performance [57]. To compare the simulation time of the proposed fault simulator with other techniques, a D-type flip-flop and a synchronous 8-bit counter are designed. These designs are then simulated along with their testbenches using saboteur [60], mutant [118], simulation command based approach [110] and the technique employed in the proposed fault simulator for fault injection. The following cases were simulated:

Case-1: D-type flip flop for 5 SEUs in 1000 clock cycles,

Case-2: Synchronous 8-bit counter for 5 SEUs in 1000 clock cycles,

Case-3: D-type flip flop for 5 stuck-at-0 faults in 1000 clock cycles, and

Case-4: Synchronous 8-bit counter for 5 stuck-at-0 faults in 1000 clock cycles.

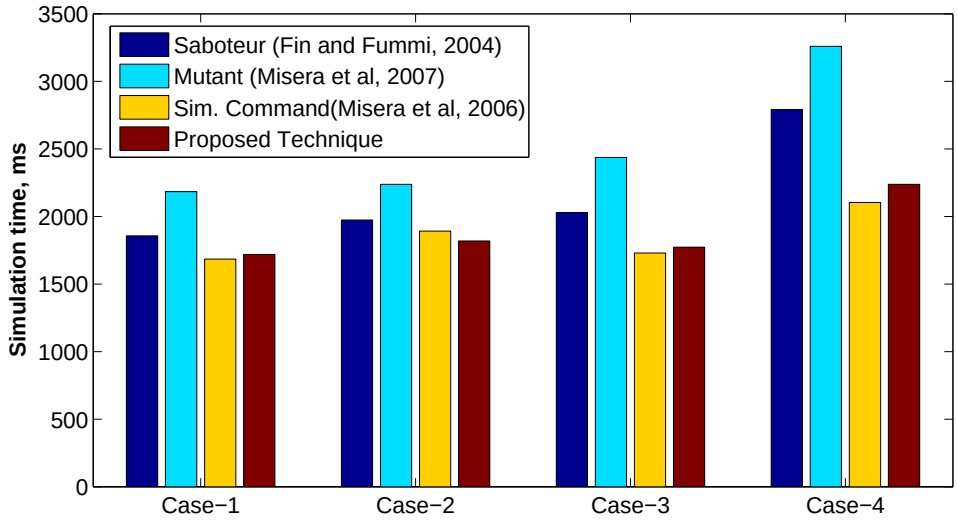


FIGURE 3.13: Comparison of simulation speed for fault simulation techniques

The simulation times for different cases are recorded as average elapsed time in ModelSim 6.2g on a Intel(R) Core(TM)2 CPU T7200 at 2GHz PC and are shown in Figure 3.13. As can be seen, the technique employed in the proposed simulator gives less simulation times than saboteur [60] and mutant-based fault injection techniques by on average 12.76% and 25.42%, respectively for the four test cases. The higher simulation time for saboteur and mutant-based fault injection is due to insertion of additional signal drivers and inflated design description (Figure 3.12(a) and (b)). The technique employed in the proposed simulator and simulation command based approach [110] have comparable simulation times with only 1.75% higher simulation time on average for the technique implemented on a prototype fault simulator. The higher simulation time in the fault

injection simulator is due to the fact that it employs iterative fault location search within the fault locations database (Figure 3.10).

3.3.4 Summary of Comparisons

The proposed fault injection simulator employs minimum intrusive design modification compared to saboteur [60], mutant [118] or simulation command based approach [110] (Section 3.3.1). Minimum design modification is achieved through replacement of the original variable and signal types to the fault injection enabler types in the design description of the DUT (Table 3.1). Using such replacements all variable or signal types in the DUT can be subjected to fault injection. As such the proposed fault injection simulator can achieve high fault representation and simulate different fault injection scenarios with various types (Section 3.3.2). High fault representation in the fault simulator is also achieved with low simulation time (by 12.76% and 25.42%) compared to saboteur [60] and mutant-based [118] approach. Also, compared to the recently reported SystemC simulation command based approach [110], the proposed fault injection simulator has comparable simulation speed (with 1.75% higher simulation time, Section 3.3.3).

3.4 MPEG-2 Decoder Case Study

In this section, an MPEG-2 decoder setup is presented as a case study to validate the effectiveness of the fault injection simulator (Section 3.2). Single-event upset (SEU) is used as fault model for validation as it is by far the most popular transient fault model used in the literature [62, 121]. The SEU injection is carried out using a Poisson distribution for fault locations within the register space (formed by the fault locations database, Section 3.2.1). Due to *sc_logic* only type restrictions, such validation is not feasible using the fault injection technique proposed in [110] (Section 3.1.4).

For validation, a behavioural SystemC model for MPEG-2 video decoder is developed based on the specification described in [124]. Figure 3.14 shows the block diagram of the MPEG-2 video decoder setup with the functional blocks (refer to Appendix A for further details to MPEG-2 video decoder). Fault injection in the decoder setup is enabled through replacement of the original variable and signal data types by the fault injection enabler types. Such type replacements create a fault locations database, where faults can be injected (Section 3.2). To incorporate timing information for fault injection in the register space, the clock of the decoder is connected to the fault injection simulator (Figure 3.14). The information related to fault injection rate is fed through the fault policy manager (FPM). With timing and fault policy information, the proposed simulator injects SEUs in the register space through the fault injection manager (Section 3.2.3). To demonstrate how the proposed fault injection simulator enables fault injection in

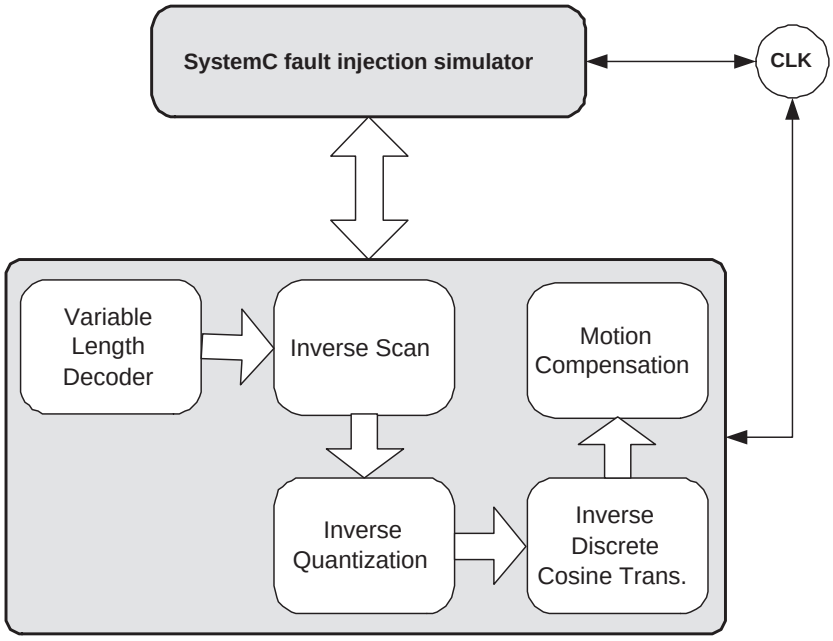


FIGURE 3.14: MPEG-2 video decoder setup for fault injection

the decoder, Figure 3.15 shows an example description of the inverse discrete cosine transformation (IDCT) functions used in the decoder (Figure 3.14). The modifications used to enable fault injection employing the proposed fault injection simulator are emboldened. As can be seen, the description includes *FIReg.h* SystemC header file, which enables the usage of different fault injection enabler types within the description (line 2). For example, the fault injection type *Reg<short>* in place of the original variable type *short* (lines 9, 11-12, 14-15) includes the variable registers *block*, *iclip*, *iclp* and *blk* in the fault locations database. Similarly, the fault injection type *Reg<int>* in place of the original variable type *int* takes control of other registers for fault injection (lines 18, 38, 63 and 71). The fault injection enabler types are transparently implemented with the functionality of the original types such that the different mathematical or logical operations within the functions (such as IDCT by row, IDCT by column) can be carried out without any further modification of the design description (Section 3.15). Using such variable replacements throughout the MPEG-2 decoder description keeps the design modifications at minimum level. Since all the variables in the functions are replaced, fault injection can be carried out with high fault representation in different registers (Section 3.3.2).

For fault injection setup in the MPEG-2 video decoder, a video sequence of 3 frames with frame size of (176 × 144) pixels is decoded (Figure 3.14). Table 3.3 shows the different inputs to the fault policy manager and outputs from the fault injection monitor of the simulator (Figure 3.5). As can be seen, a fault rate of p_B=1 × 10^{-8} is used in the fault policy manager (FPM) module. The fault rate is chosen arbitrarily for probabilistic fault

```
1: #include "config.h"
2: #include  "fim/FI Reg.h"
3: #define W1 2841 /* 2048*sqrt(2)*cos(1*pi/16) */
4: #define W2 2676 /* 2048*sqrt(2)*cos(2*pi/16) */
5: ......other constants
6:
7: /* global declarations */
8: void Initialize_Fast_IDCT _ANSI_ARGS_((void));
9: void Fast_IDCT _ANSI_ARGS_((   Reg<short> *block));
10:/* private data */
11:static  Reg<short> iclip[1024]; /* clipping table */
12:static  Reg<short> *iclp;
13:/* private prototypes */
14:static void idctrow _ANSI_ARGS_((   Reg<short> *blk));
15:static void idctcol _ANSI_ARGS_((   Reg<short> *blk));
16:
17:static void idctrow(  Reg<short> *blk){
18:   Reg<int> x0, x1, x2, x3, x4, x5, x6, x7, x8;
19:   if (!((x1=blk[4]<<11)|(x2=blk[6])|(x3=blk[2])|(x4=blk[1])
20:     |(x5=blk[7])|(x6=blk[5])|(x7=blk[3]))){
21:     blk[0]=blk[1]=blk[2]=blk[3]=.....=blk[7]=blk[0]<<3;
22:     return;
23:   }
24:   x0 = (blk[0]<<11) + 128; /* for rounding in 4th stage */
25:   /* first stage */ wait();
26:   x8 = W7*(x4+x5);   x4 = x8 + (W1-W7)*x4;
27:   x5 = x8 - (W1+W7)*x5;   x8 = W3*(x6+x7);
28:   x6 = x8 - (W3-W5)*x6;   x7 = x8 - (W3+W5)*x7;
29:   /* second stage */ wait();
30:   ......
31:   /* third stage */ wait();
32:   ......
33:   /* fourth stage */ wait();
34:   ......
35:}
36:
37:static void idctcol(  Reg<short> *blk){
38:   Reg<int> x0, x1, x2, x3, x4, x5, x6, x7, x8;
39:   if (!((x1=(blk[8*4]<<8))|(x2=blk[8*6])|(x3=blk[8*2])
40:      |(x4=blk[8*1])|(x5=blk[8*7])|(x6=blk[8*5])|(x7=blk[8*3])))
41:   {
42:     blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=....=
43:       iclp[(blk[8*0]+32)>>6];
44:     return;
45:   }
46:   x0 = (blk[8*0]<<8) + 8192;
47:   /* first stage */ wait();
48:   x8 = W7*(x4+x5) + 4;   x4 = (x8+(W1-W7)*x4)>>3;
49:   x5 = (x8-(W1+W7)*x5)>>3;   x8 = W3*(x6+x7) + 4;
50:   x6 = (x8-(W3-W5)*x6)>>3;   x7 = (x8-(W3+W5)*x7)>>3;
51:   /* second stage */ wait();
52:   ......
53:   /* third stage */ wait();
54:   ......
55:   /* fourth stage */ wait();
56:   ......
57:}
58:
59:/* 2-D inverse discrete cosine transform */
60:void Fast_IDCT(  Reg<short> *block){
61:   Reg<int> i;
62:   for (i=0; i<8; i++)
63:     idctrow(block+8*i); wait();
64:   for (i=0; i<8; i++)
65:     idctcol(block+i); wait();
66:}
67:
68:void Initialize_Fast_IDCT(){
69:   Reg<int> i;
70:   iclp = iclip+512;
71:   for (i= -512; i<512; i++){
72:     iclp[i]=(i<-256)?-256:((i>255)?255:i); wait();}
73:}
```

SystemC fault injection simulator library

Global variables and functions with fault injection enabler types

IDCT by row function with fault injection enabler types

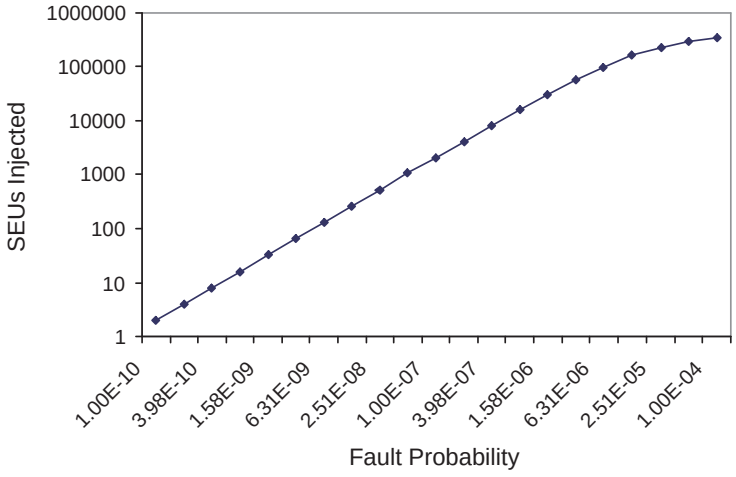
IDCT by column function with fault injection enabler types

IDCT functions and initialisations with fault injection enabler types

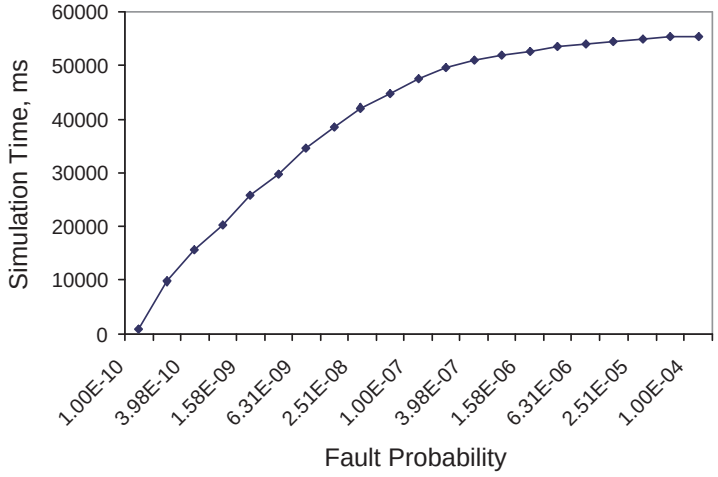
FIGURE 3.15: Example description of inverse discrete cosine transformation (IDCT) functions in the MPEG-2 decoder highlighting the modifications for enabling fault injection in the proposed fault injection simulator

Fault Policy Manager	Fault type	Single-event upset (SEU)
	Fault probability or rate	p_B=1 × 10^{-8}
	Fault probability type	Poisson's
Fault Injection Monitor	Execution time, T_E (cycles)	419287
	Register usage, R (bits/cycle)	49324
	Total faults injected, Γ	211

TABLE 3.3: MPEG-2 video decoder setup with input in the fault policy manager and output from the fault injection monitor



(a)



(b)

FIGURE 3.16: (a) Total number of faults injected for varying fault probabilities, and (b) simulation times (in ms) for varying fault probabilities of the MPEG-2 video decoder fault injection setup

injection with Poisson's distribution for determining location of SEUs within the register space (Table 3.3). Due to automatic insertion and update, the size of register space varies (Section 3.2.1). The fault injection monitor returns the average register usage of R=49kbits per cycle over the execution time of T_E=419287 clock cycles (Table 3.3). The decoder experiences Γ=211 SEUs over this time as recorded by the fault injection monitor (can also be analytically found through (3.3)). The proposed fault injection simulator can represent faults in different register types, including the variables and signals. However, an approximate total of 768 bits of the constants within pre-processor directives can not be replaced by any equivalent types using the proposed fault simulator. Assuming the total register usage is made up of these constants and the register usage (49kbits per cycle) returned by the fault injection monitor, about 98.5% registers are represented within the MPEG-2 video decoder setup for fault injection. Such high fault representation is possible due to replacement of the original types to the fault injection enabler types and their transparent implementation (Section 3.2.1).

Figure 3.16(a) shows the total number of SEUs injected recorded by the fault injection monitor for varying fault probabilities (p_B) from 10^{-10} to 10^{-4}. It can be seen that the total number of SEUs injected varies almost linearly with fault probability, p_B. For example, a total of 259 SEUs were injected at p_B=1.25×10^{-8} and 560 SEUs were injected at p_B=2.5×10^{-8}. The approximate number of SEUs injected can be found by (3.3). Figure 3.16(b) shows the corresponding simulation times as CPU elapsed time. The simulation times are recorded on an Intel Pentium-4 CPU clocked at 3.20GHz system running SystemC on RHEL 2.6.9-42. As can be seen, the simulation times (in ms) increase with higher p_B due to increased number of SEUs injected (Figure 3.16(a)). The simulation times increase almost linearly from 990 ms to 50560 ms as p_B increases from 10^{-10} to 4×10^{-7}. However, with further increase in p_B, the simulation times do not increase proportionately. This is because at higher SERs, faults injection takes place on almost every clock cycle, requiring iterative search on every cycle. To compare the faulty cases with fault-free case, further simulation is carried out without injecting any faults in the decoder. The decoder for fault free case takes only 750 ms time for decoding the same test video in 419287 clock cycles recorded from simulation compared to 990 ms for a fault probability, p_B, of 10^{-10}. The increase in the simulation time from fault-free case to faulty case is expected as fault injection requires iterative search for the fault location within the fault injection manager.

3.5 Concluding Remarks

This chapter has presented a SystemC fault injection simulator implementing a novel fault injection technique based on simulation command based approach. The fault injection in the simulator is initiated by replacement of the original variable and signal

register types with the proposed fault injection enabler types in the system description of the the device under test (DUT) (Section 3.2). In doing so, the design modifications are kept simple and less intrusive without change of functionality. Due to such replacement of various signal and variable register types, the proposed fault injection simulator benefits from high fault representation. Also, the proposed fault simulator has less simulation time than saboteur- and mutant-based approaches due to minimum design intrusion and simplicity. The simulation time of the proposed simulator is comparable to the recently reported simulation command based approach [110, 118] (Section 3.3). To demonstrate the fault injection capabilities of the fault injection simulator, an MPEG-2 video decoder setup has also been presented. It has been shown that high fault representation in variable and signal registers can be achieved using the proposed fault injection simulator (Section 3.4). This fault injection simulator is also used throughout the thesis (Chapters 4, 5 and 6) to validate the system reliability in the presence of soft errors.

Chapter 4

On-Chip Communication Architecture Comparative Analysis

A key requirement in SoC design is to find a suitable on-chip communication architecture, since the chosen architecture influences the system performance [66]. A number of on-chip communication architectures have been proposed over the years. Advanced microprocessor bus architecture (AMBA) is an industrial standard, scalable on-chip communication architecture (for brief introduction to AMBA, see Section 2.1.1, Chapter 2). Due to the demands of scalability and short time-to-market, designs of MPSoCs are being expedited towards more modular and structured on-chip communication architectures. Network-on-Chip (NoC) evolves as an emerging on-chip communication architecture with high modularity and scalability [77] (for brief introduction to NoC, see Section 2.1.2, Chapter 2). To date there has been good progress in developing flexible NoC architectures with efficient communication techniques, such as ÆTHEREAL [82], NOSTRUM [83, 84] and Intel 80-core [125].

Several studies have highlighted comparison between NoC and shared-bus AMBA, such as [17, 20, 22]. Such comparisons have been carried out using synthetic application traffic. To understand the benefits and shortcomings of these architectures in terms of performance, further investigation is needed using real application traffic. Also, with reliability as a design challenge for today's MPSoCs, such investigation should consider the impact of choice of on-chip communication architecture on reliability (for introduction to reliability, see Section 2.3, Chapter 2). This chapter provides comparison between NoC and shared-bus AMBA through analytical and simulation approach using MPEG-2 video decoder as a case study. Using cycle-accurate realistic simulations, different metrics are defined and evaluated to find out how these on-chip communication architectures compare in terms of performance and reliability.

The rest of the chapter is organised as follows. Section 4.1 gives a review of related works carried out to compare NoC with other architectures using different techniques. In Section 4.2, MPEG-2 video decoder designs with AMBA- and NoC-based on-chip architecture are presented. Different metrics are defined and evaluated towards MPEG-based comparison between NoC and AMBA in Section 4.3. Using these metrics, Section 4.4 compares the MPEG-based performance and Section 4.5 compares reliability between NoC and AMBA. Finally, Section 4.6 concludes the chapter.

4.1 Related Works

The concept of NoC derives from wired networks and was introduced in [126] showing a networked message passing architecture for multiprocessing. Further details of NoC concepts and its advantages were presented in [79] arguing that traditional shared-bus would not meet the performance requirements of future MPSoCs. It argued that NoCs would outperform shared-bus systems with better throughput and lower latency. Similar arguments for employing NoC in place of shared-bus SoC architecture has been presented in [38, 80] showing that NoC can provide well-structured on-chip communication, simplify the layout and provide with high scalability for system integration. It is demonstrated that since communication channels in NoC are multiplexed across various connecting nodes, the bandwidth, utilisation and performance are enhanced for multiprocessing. However, laying out more wires for multiplexing communication channels in NoC has been shown to cause area overhead. For example, in [33, 38] it has been shown mesh-based NoC implementation gives 6.6% area overhead compared to traditional shared-bus on-chip communication architectures. A number of other studies have also been reported showing various intuitive comparisons and design challenges using NoC. For example, in [34, 80] it is shown that future MPSoC designs would be driven by considerations like scalability, energy-efficiency, and design productivity of the on-chip communication architecture. The papers also highlighted specific problems that need to be addressed for NoC-based on-chip communication architectures, including reliability, task mapping and architecture allocation (allocation of number of processing cores and their types). Different aspects and challenges of NoC communication protocols and layering have also been extensively investigated and compared with traditional design methodologies in [81].

To date good progress has been made in the research and development of efficient NoC architectures and on-chip communication techniques. For example, ÆTHEREAL NoC architecture has been proposed by [82] with guaranteed communication services and NOSTRUM NoC architecture with layered communication approach has been proposed by [83, 84]. Recently, mesh-based NoC architecture, Intel 80-core has also been presented with clock frequency higher than 4GHz [125]. Efficient routing and communi-

cation techniques for NoC have been proposed by [43, 45] and adaptive NoC design and implementations are presented in [127]. A detailed survey highlighting the progress of NoC research and development has been presented in [19]. It also identifies general trends in design and development of NoC architecture and communication techniques. Furthermore, a set of case studies with industrial and academic prototyping and implementation, such as xPIPES [87], MANGO [86], have also been presented in this paper.

Apart from the intuitive analysis, design and developments of AMBA and NoC discussed above, a number of comparative studies based on analytical and simulation have recently been reported. A performance analyses between AMBA and NoC using single core on-chip communication has been presented in [128] using MPEG-4 application traffic. The paper identifies that existing on-chip SoC buses have limitation on data traffic bandwidth since a large number of silicon intellectual properties (IPs) share the bus. Using experimental setup for AMBA and NoC, it shows that the performance of the MPEG-4 video codec based on NoC is improved over 50% compared to shared AMBA. In another study, quantitative cost analyses and comparison of area, power, frequency, throughput, latency and energy of NoC and bus-based architectures are presented in [20, 22]. The analyses and comparisons in [22] include different bus architectures and NoC, showing different cost functions related to performance issues. An industrial comparative study highlighting the comparisons between NoC and AMBA has been presented in [20]. It was shown that NoCs have higher maximum frequency and higher throughput than shared-bus (above 750MHz compared to 250MHz for shared-bus and 100GBps compared to 5GBps for bus) due to capacitive loading in shared-bus. A review of guiding principles towards the evolution of NoC as an emerging SoC communication architecture is presented in [17]. The paper shows that NoC has higher interconnect utilisation and concurrency due to shared channels among communicating nodes compared to a shared-bus architecture. A comparative evaluation between point-to-point (P2P) and NoC with MPEG-2 video encoder has been carried out in [33] considering area, power, data parallelism, MPEG frame rate and scalability. The comparison in [33] showed that NoC has higher scalability but comparable performance as point-to-point (P2P) systems (for details of different on-chip communication architectures, see Section 2.1, Chapter 2).

Reliability in the presence of soft errors is an emerging design challenge for MPSoCs, particularly due to exacerbation of single-event upsets (SEUs) with technology scaling [123]. A number of studies have shown different fault tolerant on-chip communication architectures and techniques for MPSoCs. For example, in [129] an investigation into reliability of different NoC architectures has been reported. Based on the investigation, effective fault tolerance techniques have been proposed for different NoC configurations to mitigate the impact of soft errors. Another reliability analysis of on-chip communication architectures from performance, reliability and energy perspective has been carried out in [130]. Using such analysis an array of different fault tolerance techniques have been introduced at architectural- and algorithmic-level to tackle the reliability issues of

communication components. In [15] a fault tolerant design of interconnects in on-chip communication architectures has been considered explaining conflicting design trade-offs between reliability and performance. The impact of power minimization on reliability has been examined in [100] showing effective power-aware fault tolerance design techniques for on-chip communication architectures. Several other techniques, such as stochastic communication [131] and routing [132], have also been proposed to incorporate fault tolerance in on-chip communication architectures.

Most of the performance comparisons reported between NoC and shared-bus AMBA, such as [17, 20, 22], use synthetic traffic patterns. Other comparative studies, such as [128, 133], do not show architectural implications on application performance. Although good progress has been made in the development of reliable architectures and techniques, currently there is a lack of analysis of how on-chip communication architecture affects the reliability of MPSoCs in the presence of soft errors. For the NoC methodology to gain further maturity, such insightful analysis of reliability need to be performed highlighting comparison between dominant shared-bus AMBA and NoC using real application traffic.

4.2 Design Space

In this section, design of NoC and shared-bus AMBA architectures with MPEG-2 video decoder cores used in this work is explained briefly.

4.2.1 MPEG-2 Video Decoder Cores

MPEG-2 video decoder constitutes a major component of the current and future multimedia systems. In this work, MPEG-2 video decoder has been used as a case study for *real application* traffic (refer to Appendix A for further details regarding MPEG-2 video decoder). Figure 4.1 shows the block diagram of MPEG-2 video decoder MPSoC with four processing cores used in this work (also used in Chapter 5). The application partitioning and task mapping are carried out arbitrarily. The variable length decoder (VLD) core reads the original video bitstream from local memory, buffers and organises them in 32-bit payload data transaction units (DTUs: 32-bit payload packets for NoC or 32-bit data per transaction in AMBA). The core VLD also checks for headers and semantics to decode the bitstream into two organised video structures: header sequence and video sequence. The header sequence with the quantisation matrices and coded macroblocks (MBs) are sent to inverse scanner and quantizer (ISQ) core. The other header and video sequence specific information are sent to the motion compensator (MC) core. The ISQ core transforms the quantisation matrices and pixel blocks into two dimensional structure (i.e. $64 = 8 \times 8$). Later this is inverse quantised in the core ISQ to form discrete

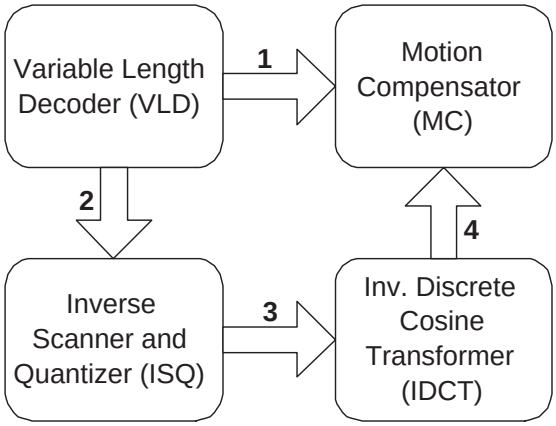


FIGURE 4.1: Block diagram of MPEG-2 video decoder

cosine transform (DCT) coefficients. The inverse discrete cosine transformer (IDCT) core transforms the two dimensional blocks into actual time domain picture-ready format in a lossy manner using the DCT coefficients. The picture-ready blocks are then sent to motion compensator (MC) core, which reads the header sequence with different semantics from core VLD and forms predictions with the decoded picture-ready video blocks from core IDCT. The decoded and motion compensated video frames are then stored in memory.

Figure 4.2 shows a simplified block diagram of a processing core used in MPEG-2 video decoder (Figure 4.1). Each processing core has a dedicated local memory of 256 kbytes, chosen to give high availability of data in the processing cores. The memory is directly connected to the input port by memory access controller. Incoming DTUs are interfaced through input channel, which is directly connected with the memory access controller. Outgoing DTUs are communicated through the output channel, which is directly connected with the processor. Optional control signals, such as *busy* and *request* signals are also connected for compatibility with different communication architectures. The signal *request* indicates the intention of the connecting module to transmit data or transmission request, while the signal *busy* communicates that the status (BUSY or NOTBUSY) of the processing cores to the connecting modules.

4.2.2 Shared-bus AMBA Design

The AMBA protocol is an open standard, on-chip bus specification that details a strategy for the interconnection and management of functional blocks within a system-on-chip (SoC) [18, 27] (see Section 2.1.1, Chapter 1 for further details). AMBA uses a set of signals connected with all other communicating modules, called the *bus* as the main interconnection unit among the masters and slaves. In this work, AMBA AHB has been used as the shared-bus architecture due to its high performance and high clock-frequency [18]. AMBA-AHB can be used to form topologies with different bus layouts

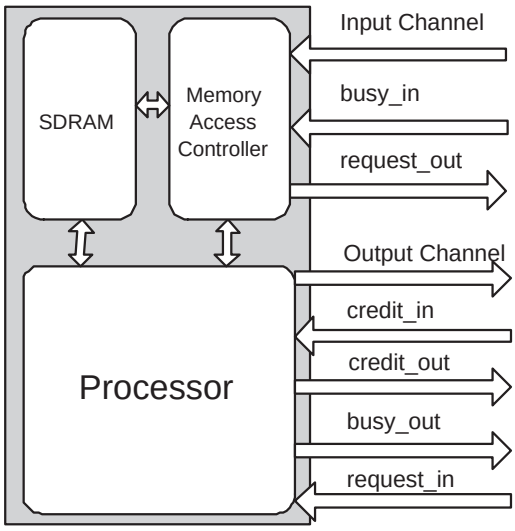


FIGURE 4.2: Simplified block diagram of a processing core used in the MPEG-2 video decoder (Figure 4.1)

with multiple layers and segments. In this work single-layer shared-AHB is used, which is one of the most commonly used shared-bus architecture to date [28].

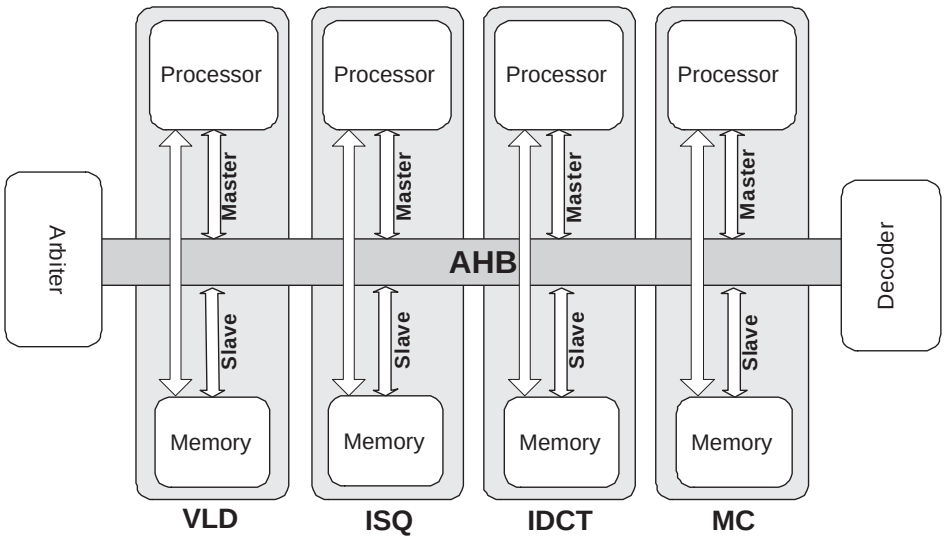


FIGURE 4.3: Block diagram of shared-bus AMBA with MPEG cores used for comparison

A block diagram of shared-bus AMBA employing MPEG-2 decoder cores used in this work is shown in Figure 4.3. The MPEG-2 video decoder cores (Figure 4.1) are configured for AMBA by using each input port as slave port and each output port as master port. Single burst sequential transfers without waiting states and 32-bit payload of each DTU are used. Bus accesses are shared and switched among the cores in the sequence of cores VLD, ISQ and IDCT until their executions are completed. Further details related to simulation setup for shared-bus AMBA is discussed in Section 4.3.1.

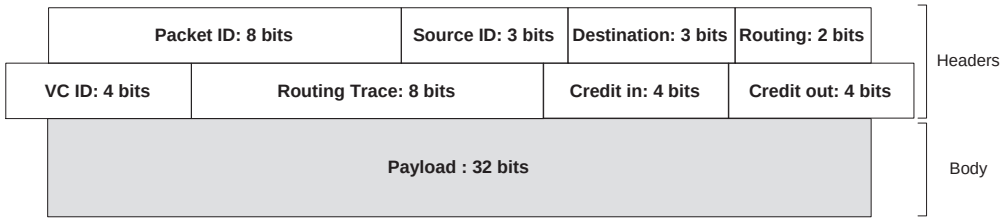


FIGURE 4.4: Packet structure used in mesh-based NoC architecture used for comparison

4.2.3 Network-on-Chip Design

A number of general purpose and application-specific NoC architectures have been proposed over the years [134, 135]. Even though application-specific NoCs outperform general purpose NoCs, the designs of such NoCs vary depending on the application, partitioning among multiple cores and resource allocations [136, 137]. The aim of this work is to compare the performance without restricting the architecture to the application itself. Hence, a general purpose architecture for NoCs is preferred. Due to its simplicity and scalability [37], mesh-based topology with XY routing is considered. Such routing provides shortest path deterministic routing between two given nodes. The floor mapping of cores has been done with shortest path between communicating cores to give minimum latency [138]. Also, single-flit (flow control unit) packet-based wormhole routing is used, which is also used in [139] (for brief introduction to communication techniques in NoC, see Section 2.1.2, Chapter 2).

NoC employs packet-based communication, where each packet is formed of two different parts: packet header and body. The packet header contains packet size and identifiers, while the body of the packet contains the actual data or payload. Figure 4.4 shows a simplified packet structure used in this work. As shown, the packets are identified by packet identification numbers (8-bits). In this NoC implementation, a circular numbering for NoC packets is used (i.e. after the highest number 2^8 is reached, the number 0 follows). For mesh-based NoC implementation, source and destination tiles are identified by 3-bits each. Other communication specific header information, such as routing type, trace, virtual channel (VC) identification and control information are also included in packet header. The size of each NoC packet used in this work is 68 bits (Figure 4.4).

The basic topological element of the NoC structure is the tile, which has three major parts: processing element (PE), network interface (NI) and switch. The MPEG-2 decoder cores are plugged into the PE and responsible for computation. The switches carry out the communication tasks, while the network interfaces (NIs) carry out the necessary decoupling between communication and computation providing packet-based communication between switches.

Every switch in an NoC has five input and five output ports and credit information to

and from each port. The switch architecture compatible for store-and-forward (SaF) or wormhole routing, which is used in this work, is shown in Figure 4.5. Each transaction is initiated with a pair of handshake signals: *busy* and *request* signals. Credit signals convey the current status of the virtual channel (VC) buffers and enable the *wormhole* routing technique. Each input channel also has buffers to store the packets as they arrive. The input channels in the NoC implementation used in this work are designed with buffer for eight packets (chosen to give high availability and bandwidth). Due to handshaking *busy* and *request* signals, channels give no congestion for pipelined transactions. Once a transaction is agreed, data arrives at the input channel of one of the ports of the switch. The VC allocator allocates the virtual channel for an incoming packet for wormhole routing, but for SaF packet routing technique, the allocation is simply limited to one virtual channel. Packets from different directions arrive at a switch controller and are served in a round-robin fashion so that packets from different ports have fair waiting times. The router decides for the outgoing port for a packet depending on the routing algorithm suggested in the packet header. In this work, XY routing is used for packets (Section 4.2.3).

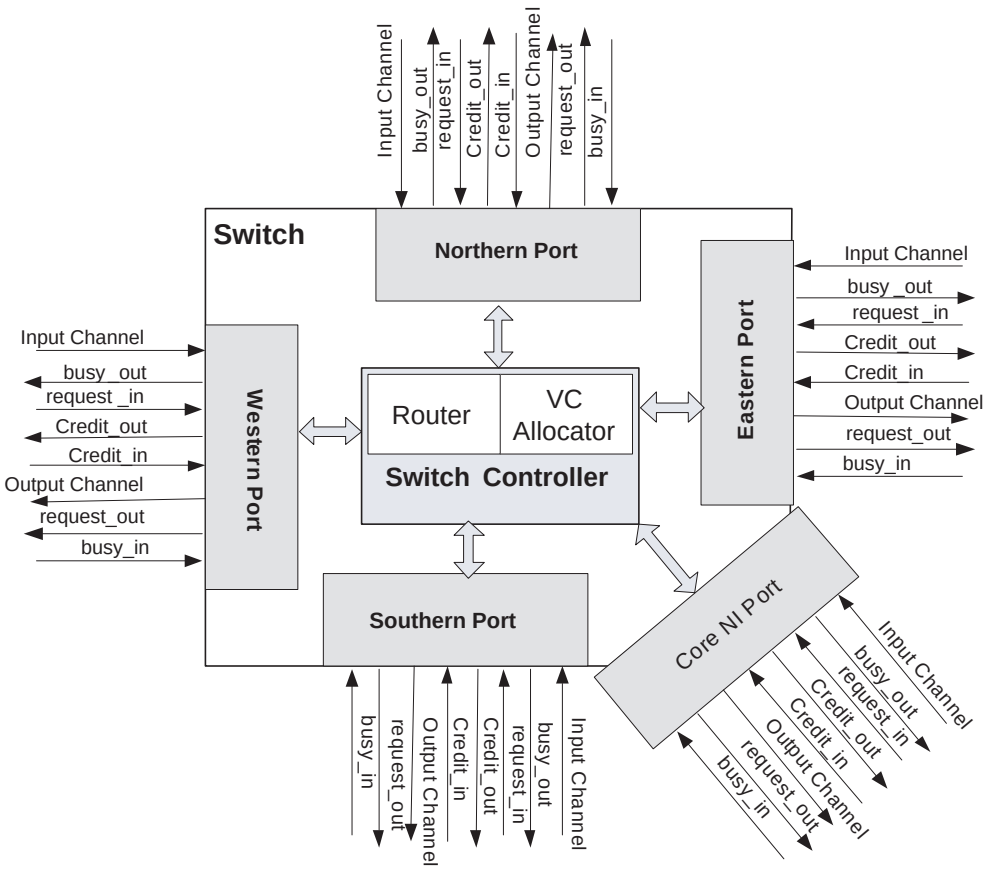


FIGURE 4.5: Switch structure for mesh-topology NoC used for comparison

The NI is an important element of NoC since it enables packet-based on-chip communication between switches. This is done by adding packet-specific information for

data communication from processing core to switch (called packetisation) and removing them for data communication from switch to processing core (called de-packetisation). A block diagram of NI module for the NoC architecture used in this work is shown in Figure 4.6. The first-in first-out (FIFO) controllers contain the FIFO memory and con-

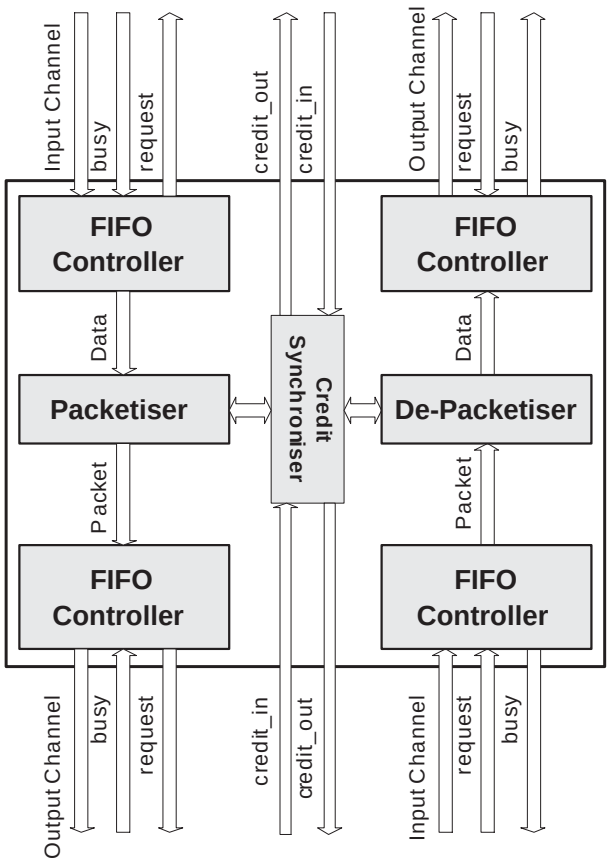


FIGURE 4.6: Block diagram of network interface of NoC used for comparison

trols the incoming data or packet from the core or switch using *busy* and *request* signals. Data from the core arrive at the FIFO memory and are packetised by the *packetiser* unit in the NI and later stored in the output FIFO memory as packet ready to be sent to the connecting switch. In similar way, incoming packets from switch are stored in the FIFO memory (Figure 4.6) and later de-packetised and stored in the output data FIFO memory to send data to the core. Credit synchroniser conveys the credit information between core and switch with programmed delays, such that the *credit_in* and *credit_out* signals are synchronised with data at the core or packet at the switch, respectively.

Figure 4.7 shows a block diagram of mesh-based (2×2) NoC used in this work employing XY routing and shortest path mapping among the communicating MPEG-2 cores (Figure 4.1). Since dedicated interconnects are laid out between processing cores, the communication in NoC is non-blocking. i.e. as long as the handshaking signals (*busy* and *request*) allow the transactions are carried out between components without blocking states. As can be seen, the NI acts as the interface between switch and processing

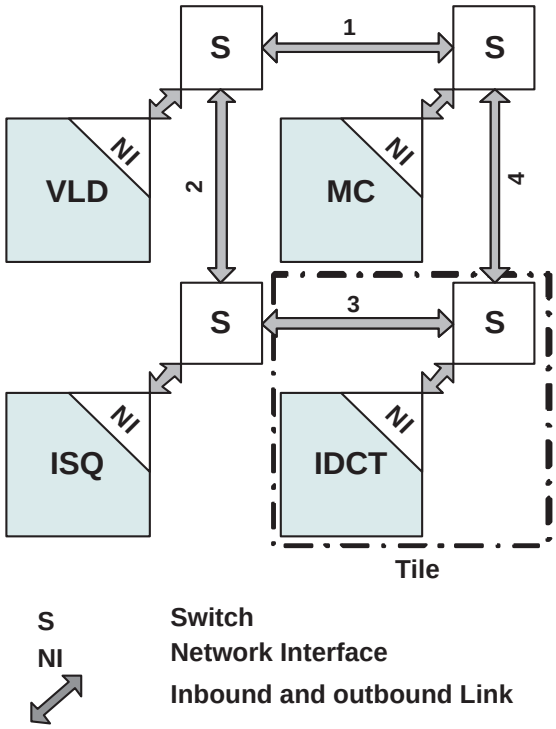


FIGURE 4.7: Block diagram of mesh-based (2 × 2) NoC with MPEG-2 video decoder cores (Figure 4.1) used for comparison

cores to enable packet based communication between switches and the switches carry out packet-based inter-core communication. The processing cores can performs necessary decoder tasks with data from the local memory and can initiate data transfer to next processing core (Figure 4.7).

4.3 Simulation Setup and Comparisons

In this section, simulation setup for NoC and shared-bus AMBA are explained. Later, comparative analyses of NoC and AMBA are presented using different metrics. The metrics are then used for comparison of application-specific performance and reliability in Section 4.4 and 4.5, respectively.

4.3.1 Simulation Environment and Test Cases

To compare the MPEG-based performance between NoC and AMBA, two separate SystemC simulators were used. To facilitate NoC related simulations, NIRGAM (NoC Interconnect Routing and Application Modelling) has been used. NIRGAM is a SystemC-based discrete-event, cycle-accurate simulator developed at the University of Southampton and provides with substantial support to experiment with NoC design in terms of

routing algorithms and applications with various topologies. NIRGAM uses behavioural modelling throughout with plug-in support for processing cores. Using separate monitor modules, it can accurately profile NoC computation and communication performance. More details on the simulator can be found in Section B.1, Appendix B. For AMBA, another simulator was used using Synopsys Designware SystemC libraries [67] in Cocentric System Studio environment (see Section B.4, Appendix B for further details). To make the simulations unbiased between the simulators, key details of the simulations setup are explained next.

1. **Core Configuration:** The processing cores are developed using SystemC behavioural modelling. Cores were plugged between the architectures without change in the application code. In NoC, NI does the necessary translations for packet-based communication (Figure 4.7), while in shared-bus AMBA, the input and output channel from the PE shown in Figure 4.2 were connected to slave input and master output port, respectively (Figure 4.3). The optional communication signals were not used in shared-bus AMBA implementation.

2. **Traffic Generation:** The processing and output traffic generation of the cores were performed core-by-core and channel-by-channel basis. Thus, the traffic generation is deterministic for a given video bitstream and application mapping [17]. No extra wait states were used in both NoC and AMBA implementations. The data injection rate is one DTU (i.e. one packet for NoC, 32-bit data for AMBA) per clock cycle. Pipelined transactions are used between the architectures.

3. **Simulation Monitoring:** To monitor and record the performance metrics, separate monitor modules were implemented on each core adding simulation-specific information to packets that are being used in inter-core communications. For AMBA, a separate monitor module was connected as a master to investigate the performance metrics on shared-bus.

4. **Interconnect Sharing:** The interconnect sharing in shared-bus AMBA is managed by designing arbiter in such a way that the master priorities are same and the interconnect sharing takes place in round robin fashion in the sequence of cores VLD, ISQ and IDCT in AMBA implementation (Figure 4.3). This technique is chosen over priority-based techniques (such as [140]), since in such techniques low priority cores can often starve (i.e. may not get the interconnect share) and unfair sharing of interconnect takes place. However, the duration for which a master core can hold the interconnect access is determined by the type of data being processed. For example, all cores hold the interconnect for 8 clock cycles for header sequence processing. During video decoding, the interconnect is held by cores VLD, ISQ and IDCT for entire macroblock (the unit of video data in terms of 16×16 pixels) processing, which is typically more than than 8 DTUs for intra macroblocks and

varies depending on type of macroblock being decoded. Since dedicated channels exist between respective communicating channels in NoC, packet communication takes place as they are permissible through channels, NIs and switches.

5. **Communication Model:** The communication model used for NoC is hop-by-hop, which requires handshake between any two connecting modules. For NoC, the cores can initiate data transactions (for *write* operation) or it can react to transaction requests (for *read* operation), while switches carry out the communication-specific routines. For AMBA, any communication (*read* or *write*) is initiated by AHB masters and AHB slaves only serve the requested service. The arbiters and decoders control the access to interconnect and decode the slave address requested. Cycle-accurate *write* and *read* model transactions for NoC and AMBA without optional wait states are shown in Figures 4.8 and 4.9. As can be seen, the NoC communication depends on the two-way handshaking signals, *busy* and *request* (Figure 4.8). As such, when a transaction is agreed through handshake, a data communication is initiated. For example, when an write operation is requested through *request_out* in HI state and *busy_in* is seen at LO state (meaning the communicating component is not busy) at falling edge of system clock, output data DATA_OUT can be written. Similarly, when a read operation is requested through *request_in* in HI state and *busy_out* seen at LO state (meaning requested component is not busy) at falling edge of system clock, input data DATA_IN can be read (Figure 4.8). For AMBA, the bus master must be granted access to the shared-bus before a transfer can commence. This process is started by the master asserting a request signal to the arbiter (Figure 4.9). Then the arbiter indicates when the master will be granted use of the bus through HREADY. A granted bus master starts an AMBA AHB transfer by driving the address (HADDR) and control signals. A write data bus (HWDATA) is used to move data from the master to a slave, while a read data bus (HRDATA) is used to move data from a slave to the master. Every transfer consists of an address and control cycle and one or more cycles for the data (Figure 4.9).

For comparison purposes, four different video test bitstreams are used with different resolutions and frame rates. Table 4.1 shows the video bitstreams with their sizes and frame rates. The simulations were carried out on Intel(R) Pentium(R)-4 CPU clocked at 3.20GHz system with 1GB RAM running SystemC on RHEL 2.6.9-67. The simulation of the four videos (Table 4.1) with decoding, monitor processes, logging took about 250, 319, 421 and 543 seconds for NoC and 231, 294, 341 and 415 seconds for AMBA. The NoC simulation times were observed in NIRGAM [25] and shared-bus AMBA simulation times were observed in Cocentric SystemC Compiler [67] as average CPU elapsed time.

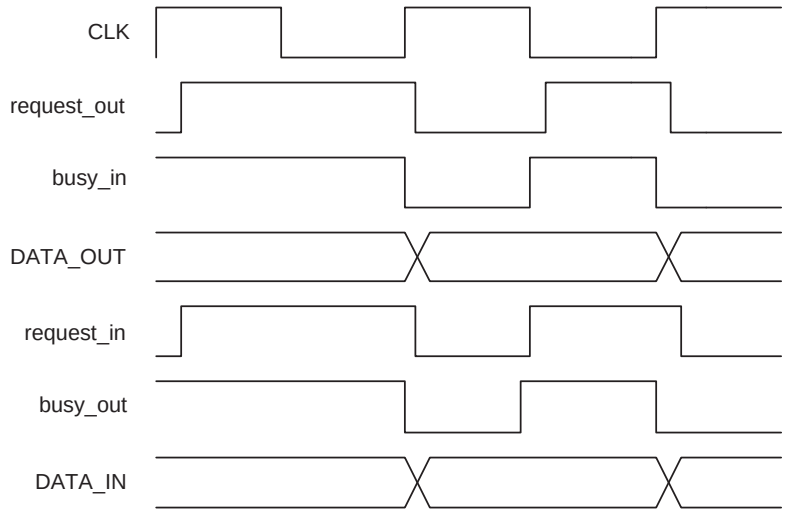


FIGURE 4.8: Cycle-accurate write and read transactions in NoC used for comparison

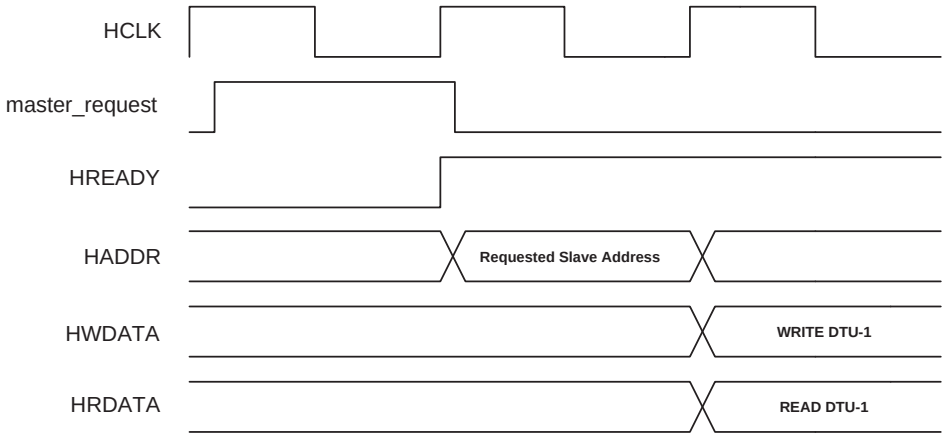


FIGURE 4.9: Cycle-accurate write and read transactions in AMBA used for comparison

4.3.2 Performance Metrics

To better understand the underlying performance of AMBA- and NoC-based MPEG-2 video decoders (Figures 4.3 and 4.7), different metrics are defined. To understand core performance, concurrency and core efficiency are defined and to understand interconnect

Video	**Frames**	**Frame rate**	**Frame Size (pixels)**
test1.m2v (tennis)	67	29	176×120 (QCIF, NTSC)
test2.m2v (flower)	55	25	352×288 (CIF, PAL)
test3.m2v (tennis)	49	25	352×576 (2CIF, PAL)
test4.m2v (flower)	43	29	704×480 (4CIF, NTSC)

Source: ftp://ftp.tek.com/tv/test/streams/Element/

TABLE 4.1: Test video bitstreams used for comparisons between AMBA and NoC

performance, channel latency and bandwidth are defined. Using these metrics, the application performance of the decoders is compared in Section 4.4.

4.3.2.1 Concurrency

Concurrency defines the number of cores that are able to execute computation at the same time and is dependent on the way processing cores communicate with each other. Higher degree of concurrency effectively reduces the total multiprocessor execution time through overlapped executions among processing cores. The average degree of concurrency, D, is defined as

$$D = \frac{\sum_{t=1}^{T_M} C(t)}{T_M} \tag{4.1}$$

where T_M is the total multiprocessor execution time (in clock cycles) and $C(t)$ is the number of cores executing the computation at t-th clock cycle. Given, $C(t) \leq C_{max}$, where C_{max} is the total number of MPSoC cores, it can be shown that, $D_{max} = C_{max}$.

Table 4.2 shows a tabular comparison of average degree of concurrency between AMBA- and NoC-based decoders recorded from the simulation logs. The number of core executions are given in column 1 (Table 4.2). The corresponding execution times for decoding *test1.m2v* by NoC and AMBA are given in columns 2-3. Similar execution times for decoding *test2.m2v*, *test3.m2v* and *test4.m2v* are given in columns 4-9. As can be seen, $T_M = 1.84 \times 10^7$ clock cycles in AMBA-based decoder, during which the application is executed with $C = 1$ core for 3.91×10^6 clock cycles, $C = 2$ cores for 3.28×10^6 clock cycles, $C = 3$ cores for 2.29×10^6 clock cycles and $C = 4$ cores for $6.85E \times 10^5$ clock cycles for video decoding bitstream *test1.m2v* (column 3). Due to shared-bus access in AMBA-based decoder, multiprocessor execution time (T_M) is mostly made up of execution with $C = 1$, 2 and 3 cores. Using (4.1) with the application times and execution times from Table 4.2, the degree of concurrency for AMBA (D_{AMBA}) is given as 1.09 for decoding *test1.m2v*. On the other hand, due to dedicated links among cores, NoC-based decoder is able to exploit the concurrency at a higher level. As can be seen, only 2.25×10^6 clock cycles are executed with $C = 1$ in NoC-based decoder, while for $C = 2$, 3 and 4 cores the recorded execution times are 2.30×10^6, 2.41×10^6 and 1.51×10^6 clock cycles for decoding video bitstream *test1.m2v*. As a result, the multiprocessor execution time (T_M) in NoC-based decoder is mostly made up of execution with $C = 2$, $C = 3$ and 4 cores (column 2). This allows NoC to overlap execution among processing cores more efficiently and reduces T_M to only 8.04×10^6 clock cycles results in a higher degree of concurrency (D_{NoC}) of 2.50, given by (4.1).

	Test video bitstreams							
	test1.m2v		*test2.m2v*		*test3.m2v*		*test4.m2v*	
C	NoC	AMBA	NoC	AMBA	NoC	AMBA	NoC	AMBA
1	2.25E+6	3.91E+6	6.88E+6	1.20E+7	1.18E+7	2.05E+7	1.27E+7	2.21E+7
2	2.30E+6	3.28E+6	7.18E+6	1.00E+7	1.23E+7	1.71E+7	1.33E+7	1.85E+7
3	2.41E+6	2.29E+6	7.20E+6	6.99E+6	1.23E+7	1.20E+7	1.33E+7	1.29E+7
4	1.51E+6	6.85E+5	4.62E+6	2.09E+6	7.91E+6	3.58E+6	8.55E+6	3.87E+6
T_M	8.04E+6	1.84E+7	2.34E+7	5.36E+7	3.95E+7	9.31E+7	6.15E+7	1.36E+8
D	2.50	1.09	2.62	1.14	2.66	1.13	2.44	1.10

TABLE 4.2: Core concurrency of NoC and AMBA for different video bitstreams

With increasing size of video in other video bitstreams, execution times for different C values increase (Table 4.2). However, the degree of concurrency varies slightly depending on the inter- and intra-frame compression in the coded video bitstreams. For example, D_{NoC} values found by (4.1) using the execution times from Table 4.2 are 2.62, 2.66 and 2.44 for decoding video bitstreams *test2.m2v*, *test3.m2v* and *test4.m2v*, respectively (found using C and T_M values in (4.1)). Similarly, D_{AMBA} values are found as 1.14, 1.13 and 1.10 for video bitstreams *test2.m2v*, *test3.m2v* and *test4.m2v*, respectively. As expected, NoC-based decoder maintains higher D compared to AMBA exploiting higher concurrency among processing cores due to spatial multiplexing of channels among cores. On average NoC-based decoder has 2.18 times higher degree of concurrency for the given architecture (Figure 4.1) when compared to AMBA-based decoder. Clearly NoC suits MPSoC architectures, where concurrent processing is desirable.

4.3.2.2 Core Efficiency

Core efficiency defines how efficiently processing cores can utilise the computation cycles within the execution time. Total execution time of a processing core is given by the sum of processing and non-processing times as

$$T_E = T_P + T_{NP} \leq T_M, \tag{4.2}$$

where T_E is the execution time, T_M is the multiprocessor execution time, T_P is the total processing time (i.e total number of computation cycles) and T_{NP} is the total non-processing time, all in clock cycles. The non-processing time, T_{NP} is defined as

$$T_{NP} = T_R + T_W + T_I, \tag{4.3}$$

where T_R is the reading time of the input data at the processing core and T_W is the time required for output data to be written at the output ports and are both 0 clock cycle in NoC and AMBA due to local memory and pipelined communication, T_I is the idle time and is the major contributor to non-processing time. The idle time, T_I, is caused by any of the following three reasons: i) not having interconnect to the master access during writing, ii) not having enough data to process, or iii) not being able to read from memory due to blocking access through shared interconnect. To investigate quantitatively how effectively the computation cycles are being utilised within execution time of a core, core efficiency, σ, is defined using (4.2) and (4.3) as

$$\sigma = \frac{T_P}{T_E} = \frac{T_E - T_{NP}}{T_E}, \tag{4.4}$$

where T_E, T_P and T_{NP} are total core execution, computation and non-computation cycles defined by (4.2) and (4.3). Table 4.3 shows the execution times (T_E) and non-

processing times (T_{NP}) of each core for decoding four different videos recorded from the simulation logs. As shown in Table 4.3, the core VLD has 1.20×10^6 clock cycles of non-processing time (T_{NP}) and 6.43×10^6 clock cycles of execution time (T_E) for decoding video bitstream *test1.m2v* in NoC-based decoder. The non-processing times in core VLD in NoC-based decoder are caused by waiting times for having output interconnect busy or core memory full. Similar non-processing times also take place in the cores ISQ, IDCT and MC. These non-processing times take place due to waiting for DTUs to arrive for processing as core ISQ receives DTUs from core VLD, core IDCT receives DTUs from core ISQ and core MC receives DTUs from cores VLD and IDCT. Using execution times (T_E) and non-processing times (T_{NP}) from Table 4.3 in (4.4) the average core efficiencies (σ) of cores VLD, ISQ, IDCT and MC in NoC are found by as 81%, 77%, 89% and 82%, respectively. The higher σ for cores IDCT and MC are expected due to high T_E and low T_{NP} compared to the other cores. Due to shared interconnect access, idle times make up a major component of the non-processing times of all the cores in AMBA-based decoder. As a result, core VLD has higher non-processing time (T_{NP}=4.85×10^6 clock cycles) and also higher execution time (T_E=1.34×10^7 clock cycles) for decoding video bitstream *test1.m2v* in AMBA-based decoder (Section 4.3.2.1). Similarly other processing cores also experience higher non-processing times. With these high non-processing times in AMBA-based decoder, it gives lower core efficiencies (σ) of 45%, 36%, 39% and 37% for the cores VLD, ISQ, IDCT and MC, respectively (Table 4.3). Note that core VLD gives higher σ compared to other cores in AMBA. This is because core VLD reads input bitstream directly from memory for decoding header and video sequences and hence has lower non-processing times. Figure 4.10 shows a graphical comparison of the average core efficiencies (σ) of AMBA- and NoC-based decoders (Figures 4.3 and 4.2). Due to dedicated interconnects, cores in NoC utilise the execution cycles more efficiently compared to AMBA. For example, the core IDCT has 87% core efficiency (σ) on average in NoC-based decoder compared to only 39% in AMBA-based decoder.

Video	Arch.	Core VLD		Core ISQ		Core IDCT		Core MC	
		T_E	T_{NP}	T_E	T_{NP}	T_E	T_{NP}	T_E	T_{NP}
test1.m2v	NoC	6.43E+6	1.10E+6	3.78E+6	1.23E+6	6.37E+6	6.89E+5	6.69E+6	1.16E+6
	AMBA	1.34E+7	8.07E+6	7.48E+6	4.93E+6	1.37E+7	8.02E+6	1.46E+7	9.07E+6
test2.m2v	NoC	1.87E+7	2.60E+6	1.42E+7	1.61E+6	1.85E+7	1.88E+6	1.94E+7	3.38E+6
	AMBA	3.96E+7	2.35E+7	2.86E+7	1.60E+7	4.04E+7	2.38E+7	4.26E+7	2.66E+7
test3.m2v	NoC	3.23E+7	4.85E+6	2.50E+7	3.92E+6	3.20E+7	3.52E+6	3.36E+7	5.66E+6
	AMBA	6.98E+7	4.24E+7	5.14E+7	3.03E+7	7.02E+7	4.17E+7	7.40E+7	4.61E+7
test4.m2v	NoC	4.55E+7	7.29E+6	3.66E+7	5.81E+6	4.73E+7	5.47E+6	4.86E+7	9.08E+6
	AMBA	9.83E+7	6.01E+7	7.52E+7	4.44E+7	1.04E+8	6.22E+7	1.07E+8	6.75E+7

TABLE 4.3: Execution and non-processing times (in clock cycles) of processing cores in NoC and AMBA for different video bitstreams (Table 4.1)

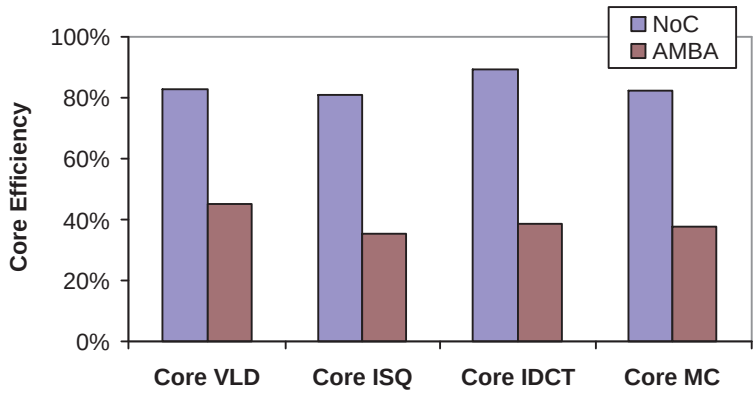


FIGURE 4.10: Average core efficiencies (σ) of AMBA and NoC MPEG-2 video decoder (Figure 4.1)

4.3.2.3 Channel Latency

Channel latency (in time units) represents an important performance parameter for on-chip interconnects. Per data transaction unit (DTU: packet with 32-bits payload for NoC, each 32-bit transaction data for AMBA) channel latency is a measure of how fast per DTU is routed over the channel from output of a processing core to input of the communicating core. For both NoC and AMBA, the pipelined transaction of data takes place in multiple hops starting from initiator core to destination core (Section 4.3.1). Considering no waiting states in NoC and AMBA, the average per DTU channel latency, L_{ch} is defined as

$$L_{ch} = \frac{1}{\mathcal{N}} \sum_{n=1}^{\mathcal{N}} \left[\tau_{c-in}^{S}(n) + \tau_{in-in}^{S-D}(n) + \tau_{in-c}^{D}(n) \right], \tag{4.5}$$

where $\tau_{c-in}^{S}(n)$ is the time elapsed for data to travel from source output port to source interconnect port, $\tau_{in-in}^{S-D}(n)$ is the time elapsed for data to travel from source interconnect port to destination interconnect port and $\tau_{in-c}^{D}(n)$ is the time elapsed for data to travel from destination port to the destination memory, all for n-th DTU out of total $\mathcal{N}$ DTUs. For AMBA, $\tau_{c-in}^{S}(n) = 1$ clock cycle after bus access is granted and locked. During $\tau_{in-in}^{S-D}(n) = 1$ clock cycle the arbiter does the necessary routing of the data and notifies the slave port. Due to direct memory interface, $\tau_{D_{in-c}}(n) = 0$ clock cycle. Minimum channel latency (without waiting states) per DTU for AMBA, given by (4.5), is $L_{ch} = 2$ clock cycles. Due to symmetric nature of NoC channels, $\tau_{D_{c-in}}(n) = \tau_{S_{in-c}}(n) = 3$ clock cycles involving intermediate NI packetising and de-packetising (Section 4.2.3). The delay, $\tau_{in-in}^{S-D}(n)$, in (4.5) involves communication over an array of switches for each DTU that travels through the channel and depends on the number of intermediate switches

travelled. Independent of routing algorithm or path travelled, $\tau_{in-in}^{S-D}(n)$ is given by

$$\tau_{in-in}^{S-D}(n) = \sum_{k=1}^{\mathcal{K}-1}[\tau_{ic-r}^{k}(n) + \tau_{r}^{k}(n) + \tau_{r-oc}^{k}(n) + \tau_{oc-ic}^{k-(k+1)}(n)]. \tag{4.6}$$

Equation (4.6) is a result of multi-hop communication through $\mathcal{K}$ intermediate switches. The time required for the n-th packet to travel from input channel to the router of the k-th switch, $\tau_{ic-r}^{k}(n)$ is 1 clock cycle. Also, the time required for routing decision on the k-th switch for n-th packet, $\tau_{r}^{k}(n)$ is 1 clock cycle. The n-th packet travels from router to the output channel of the k-th switch immediately in the NoC implementation and hence $\tau_{r-oc}^{k}(n) = 0$ clock cycle. Finally, the time required for the n-th packet to travel from output channel of k-th switch to input channel of the $(k+1)$-th switch, $\tau_{oc-ic}^{k-(k+1)}(n)$ is 1 clock cycle. Using (4.5) and (4.6), NoC has a minimum channel latency (L_{ch}) of 9 clock cycles (when $\mathcal{K} = 2$ for shortest path mapping and XY routing). The channel latency (L_{ch}) for AMBA- and NoC-based decoders (Figures 4.3 and 4.7) found through simulations are also 9 clock cycles and 2 clock cycles, respectively. For a given application traffic, channel latency sets up the major difference between the on-chip communication architectures.

4.3.2.4 Bandwidth

Bandwidth (in bits per second) is a measure of the amount of data that can be passed through the interconnect in a given period of time. According to [22], the maximum available bandwidth of a node, $BW_{arch_{MAX}}$, in any on-chip communication architecture is given by

$$BW_{arch_{MAX}} = \frac{\sum_{l=1}^{L_{arch}} w_{arch}(l) f_{arch}(l)}{H_{arch}} \ (bits/cycle), \tag{4.7}$$

where L_{arch} is the number of outgoing links being used, $w_{arch}(n)$ is the size of the l-th link in data bits only (or number of data wires), $f_{arch}(l)$ is the frequency of l-th link of the architecture being considered and H_{arch} is the number of hops (in clock cycles) required for node-to-node communication. Due to spatial multiplexing of outgoing channels in NoC, $L_{NoC} = 4$ links (Figure 4.7) and $H_{NoC} = 9$ clock cycles (Section 4.3.2.3). On the other hand, due to shared-bus access in AMBA, $L_{AMBA} = 1$ (Figure 4.3) and $H_{AMBA} = 2$ clock cycles (Section 4.3.2.3). Considering injection rate of single-packet per clock cycle in AMBA- and NoC-based decoders, the maximum bandwidth of AMBA and NoC are found as

$$BW_{NoC} = \frac{(32 \times f_{NoC})}{9} \ (bits/cycle), \ and \tag{4.8}$$

$$BW_{AMBA} = \frac{(32 \times f_{AMBA})}{(2 \times 4)} \ (bits/cycle). \tag{4.9}$$

From (4.8) and (4.9), AMBA has 9% bandwidth advantage over NoC. In practice, the actual operating frequency will also depend on capacitive loading. According to [20], due to capacitive loading and global wire lengths in AMBA, maximum operating frequency of NoC is 3 times that of AMBA, i.e. $f_{NoC} = 3 \times f_{AMBA}$. Using these maximum operating frequencies the bandwidth definitions in (4.8) and (4.9) give NoC a 2.7 times higher bandwidth advantage over AMBA.

4.4 Comparative Application Performance

Using the different parameters defined in Section 4.3.2, MPEG-based performance comparison between NoC and shared-bus AMBA is carried out in this section. The application performance is evaluated and compared between AMBA- and NoC-based decoders (Figures 4.3 and 4.7) in terms of per macroblock (i.e. block of 16×16 pixels) decoding time and operating frequency for different test video bitstreams (Table 4.1).

4.4.1 Per Macroblock Decoding Time

Given a video bitstream, the efficiency and performance of an MPEG-2 decoder is defined by the time required to decode the bitstreams. The determination of this time can be performed through average per macroblock (MB) decoding time T_{MB} (also used in [128]). Each MB in the video bitstreams (Table 4.1) is comprised of 16×16 pixels segment in a frame. The number of MBs encoded in the bitstreams vary depending on the resolution of the video frames and compression of video encoding achieved using correlation that exists within and among the frames. However after decoding, a QCIF frame has 9 rows with 11 MBs per row (*test1.m2v*), a CIF frame has 18 rows with 22 MBs per row (*test2.m2v*), a 2CIF frame has 36 rows with 22 MBs per row (*test3.m2v*) and finally a 4CIF frame has 36 rows with 44 MBs per row (*test4.m2v*, Table 4.1). The total number of macroblocks to be decoded in a video bitstream can, thus, be found by multiplying the frame size (in number of MBs) by the number of frames given in Table 4.1. To find per MB decoding time (T_{MB}) multiprocessor execution time (T_M, obtained from Table 4.2) is divided by the number of MBs per video bitstream. Table 4.4 shows T_{MB} for each video bitstream (Table 4.1) for the MPEG-2 video decoder (Figure 4.1) implemented with NoC and AMBA on-chip communication architectures (column 2 and 3). As can be seen, T_{MB} for AMBA is 2774 clock cycles for the video bitstream *test1.m2v*. This is approximately 2.30 times higher than T_{MB} for NoC, which is 1212 clock cycles. The less per macroblock processing time in NoC-based decoder is due to higher concurrency and overlap of execution of processing cores (Section 4.3.2.1). Due to low concurrency and core efficiency (Sections 4.3.2.1 and 4.3.2.2), AMBA-based decoder has large non-processing time and hence higher multiprocessor execution time (T_M), giving on average

Video	T_{MB}**, clock cycles**	
	NoC	AMBA
test1.m2v	1212	2774
test2.m2v	1074	2461
test3.m2v	1018	2398
test4.m2v	903	1990

TABLE 4.4: Per macroblock (MB) decoding time of AMBA- and NoC-based decoders for decoding the test video bitstreams (Table 4.1)

2.27 times higher T_{MB} for AMBA than for NoC. Similar comparisons are also noted with the other video bitstreams (Table 4.4). Note that for larger videos, T_{MB} decreases slightly due to higher intra-frame correlation within video frames [141].

4.4.2 Operating Clock Frequency

For decoding a given video bitstream, lower operating clock frequency is a desired performance feature of MPSoCs. This is because high clock rates dissipate higher power [20]. The operating clock frequency required to decode a given video bitstream at a specified frame rate can be found as,

$$f = (T_{MB} \times \text{MBs/frame} \times fps), \tag{4.10}$$

where fps is the specified frames per second for the video bitstreams (Table 4.1). The approximate operating clock frequencies (f) for AMBA- and NoC-based decoders found using (4.10) for decoding different video bitstreams (Table 4.1) are shown in Figure 4.11. As can be seen, due to lower T_{MB} in NoC-based decoder (Table 4.4), it can operate at lower clock frequency (f_{NoC}) than that for AMBA-based decoder (f_{AMBA}) to decode different video bitstreams. For example, for decoding video bitstream *test2.m2v*, NoC-based decoder requires only 10.6MHz compared to 24.5MHz for AMBA to achieve the frame rate of 25 fps (Table 4.1). NoC-based decoder consistently outperforms AMBA-based decoder with lower operating clock frequency than AMBA-based decoder (on average f_{NoC} is 0.43% of f_{AMBA}).

4.4.3 Impact of Architecture Allocation

Architecture allocation is a system-level design step for MPSoCs that deals with the allocation of processing elements and their interconnections in the architecture (see Section 2.4.2.1, Chapter 2 for further details). In this work, architecture allocation is referred to as the allocation of number of processing cores in the MPSoC architecture. In

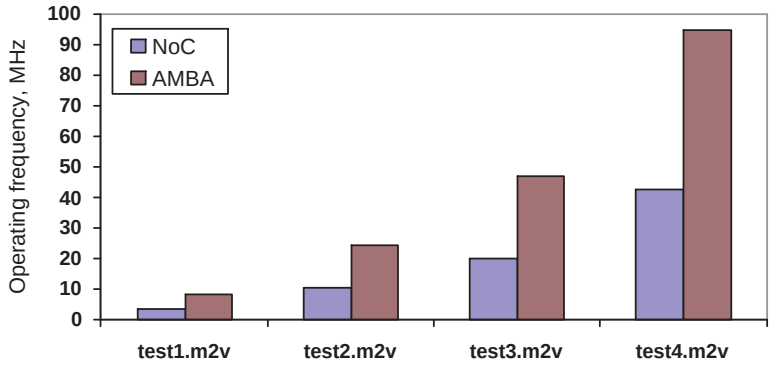


FIGURE 4.11: Required clock frequency of NoC and AMBA for decoding different test video bitstreams (Table 4.1) at specified frame rate

Sections 4.4.1 and 4.4.2, the per MB decoding time (T_M) and operating clock frequency (f) have been found using the MPG-2 decoder with four processing cores (Figure 4.1) for AMBA- and NoC-based decoders. This section demonstrates the impact of architecture allocation on the application performance.

Table 4.5 shows different architecture allocations of MPEG-2 decoder with 2, 3 and 5 cores along with mapped tasks per core (mapping of architecture with 4 cores is shown in Figure 4.1). The application task mapping is carried out arbitrarily to reflect MPSoC. As can be seen, each core is assigned a specific functionality within the decoding process, such as motion compensation, inverse scan and quantisation, inverse discrete cosine transformation, etc. (Table 4.5). Table 4.6 shows the multiprocessor execution time (T_M) and per macroblock decoding time (T_{MB}) using AMBA- and NoC-based MPEG-2 decoder for different architecture allocations (Table 4.5). The multiprocessor execution time (T_M) and per macroblock decoding time (T_{MB}) of architecture with 2 allocated cores is shown in column 3, while that of 3 and 5 cores are shown in columns 4 and 5 (Table 4.6). The corresponding T_M and T_{MB} for AMBA and NoC for different video bitstreams are shown in rows 2-5 (Table 4.6).

Allocation	**Core**	**Mapped Tasks**
2 Cores	Core 1	variable length decoding & motion compensation
	Core 2	inverse scan, quantisation & inverse discrete cosine transformation
3 Cores	Core 1	variable length decoding inverse scan and quantisation
	Core 2	motional compensation
	Core 3	inverse discrete cosine transformation
5 Cores	Core 1	variable length decoding
	Core 2	inverse scan and quantisation
	Core 3	discrete cosine transformation by row
	Core 4	discrete cosine transformation by column
	Core 5	motion compensation

TABLE 4.5: Task distribution of MPEG-2 video decoder among cores for different architecture allocations

Video	Arch.	2 Cores		3 Cores		5 Cores	
		T_M	T_{MB}	T_M	T_{MB}	T_M	T_{MB}
test1.m2v	NoC	1.48E+7	2235	1.11E+7	1680	6.86E+6	1034
	AMBA	1.83E+7	2759	1.89E+7	2855	1.82E+7	2741
test2.m2v	NoC	4.35E+7	1996	3.24E+7	1489	2.01E+7	924
	AMBA	5.34E+7	2450	5.58E+7	2563	5.30E+7	2432
test3.m2v	NoC	7.34E+7	1890	5.47E+7	1411	3.36E+7	865
	AMBA	8.93E+7	2301	9.51E+7	2451	8.81E+7	2271
test4.m2v	NoC	1.12E+8	1650	8.52E+7	1251	5.36E+7	790
	AMBA	1.39E+8	2020	1.47E+8	2160	1.42E+8	2080

TABLE 4.6: Impact of architecture allocation on the multiprocessor execution time (T_M) and per macroblock decoding time (T_{MB})

As can be seen, the NoC-based decoder outperforms AMBA-based decoder in terms of per macroblock decoding time (T_{MB}) for all architecture allocations due to reduced multiprocessor execution time (T_M). The higher multiprocessor execution time (T_M) in AMBA-based decoder is caused by shared interconnect access (Section 4.3.2.1). As a result of high T_M, AMBA-based decoder with 2 processing cores gives 24% higher per macroblock decoding time (T_{MB}), while AMBA-based decoder with 3 processing cores gives 73% higher per macroblock decoding time (T_{MB}) than NoC with similar architecture allocations (columns 3-6). Similar trends were also observed with other video bitstreams for increasing architecture allocations (Table 4.6). Note that AMBA-based decoder performs better for architecture allocation with lower number of processing cores than that with higher number of processing cores. This is because with lower number of processing cores, the waiting time for a processing core to securing the interconnect access decreases in AMBA. For example, AMBA-based decoder with 2 processing cores has a per macroblock decoding time (T_{MB}) of 2020 clock cycles, while decoder with 3 processing cores has per macroblock decoding time (T_{MB}) of 2160 clock cycles, while decoding *test4.m2v* (Table 4.5).

To demonstrate the impact of architecture allocation on the application performance in terms of operating clock frequency (f), Figure 4.12 shows the operating clock frequencies of AMBA- and NoC-based decoders found through (4.10) using the T_{MB} values in Table 4.6. As can be seen, NoC can exploit the concurrency (Section 4.3.2.1) and

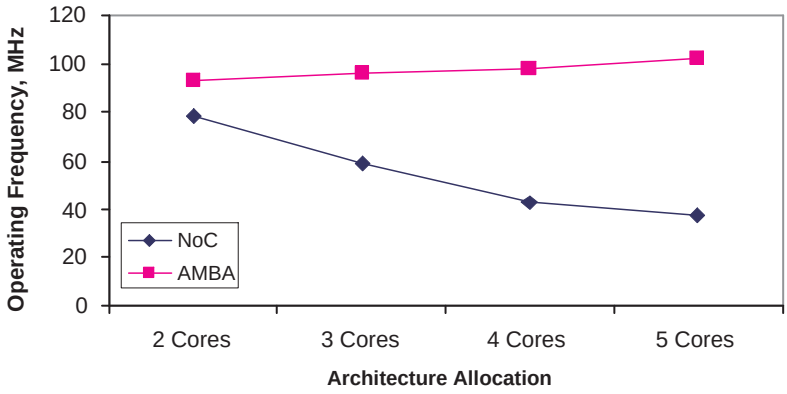


FIGURE 4.12: Operating clock frequencies of AMBA and NoC (in MHz) for decoding *test4.m2v* with different architecture allocations

higher core efficiency (Section 4.3.2.2) and effectively reduce the operating clock frequency required to decode a given video bitstream for increasing number of allocated processing cores (Figure 4.12). It is evident that as the number of cores increases, the operating frequency can scale up for AMBA due to higher T_{MB} as a result of lower core concurrency (Section 4.3.2.1) and core efficiency (Section 4.3.2.2). However, the fact that high clock frequencies in AMBA can be inhibited by increased gate delay due to capacitive loading [142], this can restrict some multiprocessor applications with many

cores to operate on AMBA. According to [20], such capacitive loading causes AMBA to a have a maximum operating frequency of 250MHz compared to 750MHz for NoC.

4.5 Comparative Reliability Analysis

An emerging challenge in today's MPSoCs is reliability in the presence of soft errors, particularly due to single-event upsets (SEUs) [15, 130] (for brief introduction to reliability, see Section 2.3, Chapter 2). In this section, reliability of the AMBA- and NoC-based decoders is studied in terms of SEUs experienced in the computation cores and communication interconnects through injection of SEUs for a given soft error rate (in SEUs per bit per clock cycle). The impact of injected SEUs is also examined at application level rather than architectural-level. The fault injection model used for reliability analysis is described next, followed by the impact of SEUs on the on-chip communication architectures in Section 4.5.2 and impact of architecture allocation on reliability in Section 4.5.3.

4.5.1 Fault Injection Model

In this work, SEU-based fault injection is carried out using the fault injection simulator proposed in Chapter 3. The injection of SEUs using this simulator is initiated through replacement of variable or signal types in the original design specification to equivalent fault injection enabler types. Such type replacement enables the formation of a fault locations database, which contains the target registers for SEU injection. The simulator injects SEUs based on the specified soft error rates and probability distribution to identify fault locations within the fault locations database. Figure 4.13 shows the fault injection setup employing the fault injection simulator used for the MPEG-2 decoder with four processing cores (Figure 4.1). Using type replacements for variable/signal in the original design specification, the simulator enables formation of five fault locations databases: one for each of the four processing cores and a centralised fault locations database for the interconnects. For a given soft error rate (SER, in number of SEUs per bit per cycle), the number of SEUs to be injected within each fault locations database is found and their locations are determined by Poisson distribution. The system clock is connected to the fault injection simulator to enable timing information for fault injection (Figure 4.13). Using simulation-specific monitor modules, total register usage and number of faults injected can be found.

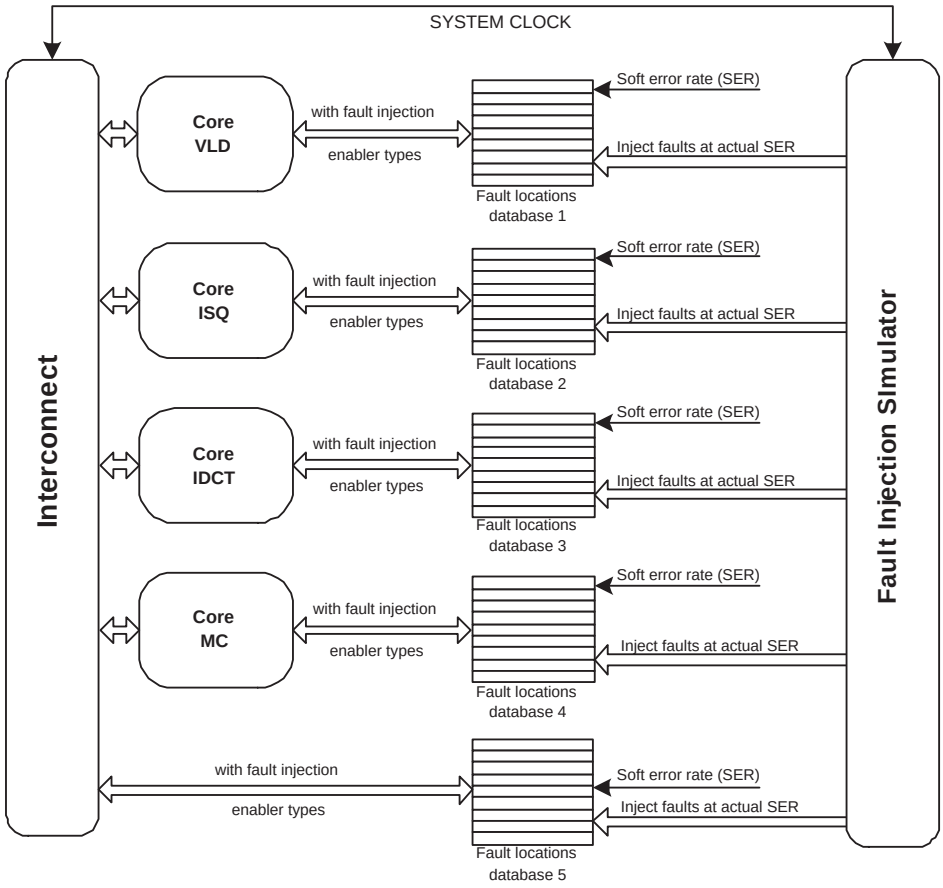


FIGURE 4.13: Fault injection setup used for comparative reliability analyses between AMBA and NoC on-chip communication architecture

4.5.2 Impact of SEUs Injected

Reliability of an application against SEUs is related to the total number of SEUs experienced over a given time [24]. The aim in this section is to analyse how the reliability of MPEG-2 video decoder is affected by the choice of AMBA and NoC on-chip communication architecture. To this end, the following investigations are carried out:

- Evaluation of the number of SEUs experienced during computation, $\mathcal{F}_{comp}$, to show how MPEG-2 computation is affected,
- Evaluation of the number of SEUs experienced during inter-core communication, $\mathcal{F}_{comm}$, to show how interconnects are affected, and
- Evaluate the impact of total SEUs, $\mathcal{F} = \mathcal{F}_{comp} + \mathcal{F}_{comm}$, at an application-level.

In the following sections, $\mathcal{F}_{comp}$ and $\mathcal{F}_{comm}$ of AMBA- and NoC-based decoders are evaluated and compared. Furthermore, the impact of $\mathcal{F}$ is examined at application-level and the impact of architecture allocation on system reliability is investigated.

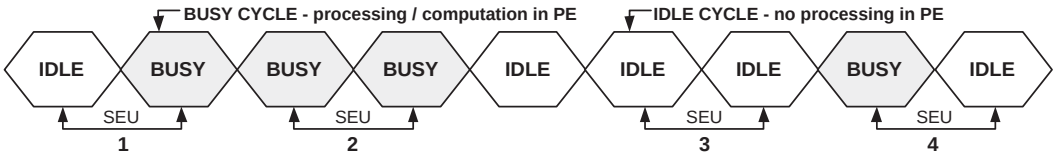


FIGURE 4.14: Manifestation of SEUs during computation cycles of a processing core

4.5.2.1 SEUs Experienced During Computation

The SEUs affect computation of a processing core through perturbation of the registers. Figure 4.14 shows how SEUs manifest themselves in registers of the processing cores during computation. As can be seen, SEUs extending between two IDLE cycles (instance 3) do not affect computation process as no computation takes place in these cycles. On the other hand, SEUs that are injected between BUSY cycles (instance 2) or between BUSY and IDLE cycles (instances 1 and 4) are likely to affect computation process (Figure 4.14). Hence, for a given soft error rate (SER), the effective number of SEUs experienced during computation ($\mathcal{F}_{comp}$) can be given as the number of SEUs experienced by the computation cycles (in instances 1, 2 and 4) during execution of a processing core. The $\mathcal{F}_{comp}$ of an MPSoC decoder with C processing cores can be given as

$$\mathcal{F}_{comp} = \sum_{i=1}^{C} \left(T_i - T_i^{I-I} \right) R_i \lambda \quad , \tag{4.11}$$

where λ is the SER (in SEUs per bit per cycle), T_i is the execution time (in clock cycles), T_i^{I-I} is the number of idle-to-idle transitions within T_i (in clock cycles) and R_i is the register usage (in bits per cycle), all for i-th processing core. The R_i gives a measure of per core register usage by the application, since SEUs in other registers have no impact [24]. The R_i is given as

$$R_i = \frac{1}{T_i} \sum_{t=1}^{T_i} R_{i,t} \quad . \tag{4.12}$$

where $R_{i,t}$ is the number of registers (in bits) used by MPEG computation process at t-th clock cycle in i-th processing core. Table 4.7 shows execution time, T_i, idle-idle transition cycles, T_i^{I-I}, and register usage, R_i, of each processing core in AMBA- (Figure 4.3) and NoC-based decoders (Figure 4.7) for decoding different video bitstreams (Table 4.1). The execution times (T_i and T_i^{I-I}) and the register usage (R_i) of AMBA and NoC-based decoder cores VLD, ISQ, IDCT and MC are shown in columns 3-6 (Table 4.7). The T_i and T_i^{I-I} values of AMBA- and NoC-based decoders are obtained from SystemC cycle-accurate simulations (Sections 4.2.2 and 4.2.3) and R_i values are found through SystemC fault simulations (Section 4.5.1).

Video	Arch.	Core VLD			Core ISQ			Core IDCT			Core MC		
		T_i, cyc. ($\times 10^6$)	T_i^{I-I}, cyc. ($\times 10^6$)	R, kb/c.	T_i, cyc. ($\times 10^6$)	T_i^{I-I}, cyc. ($\times 10^6$)	R, kb/c.	T_i, cyc. ($\times 10^6$)	T_i^{I-I}, cyc. ($\times 10^6$)	R, kb/c.	T_i, cyc. ($\times 10^6$)	T_i^{I-I}, cyc. ($\times 10^6$)	R, kb/c.
test1.m2v	NoC	6.43	0.42	23.0	3.78	0.41	19.3	6.37	0.05	19.4	6.69	0.23	25.2
	AMBA	13.4	2.4	22.5	7.48	1.9	19.0	13.7	1.4	19.1	14.6	1.6	24.7
test2.m2v	NoC	18.7	1.2	23.1	14.2	1.16	19.3	18.5	0.14	20.2	19.4	0.68	25.3
	AMBA	39.6	7.5	22.7	28.6	7.3	19.0	40.4	4.3	19.7	42.6	5.1	24.7
test3.m2v	NoC	32.3	2.2	23.4	25.0	3.0	19.4	32.0	0.25	20.5	33.6	1.2	25.5
	AMBA	69.8	14.0	22.7	51.4	13.0	19.0	70.2	7.5	19.8	74.0	9.2	24.8
test4.m2v	NoC	45.5	3.2	23.9	36.6	4.4	19.5	47.3	0.38	20.7	48.6	1.7	25.7
	AMBA	98.3	19.9	23.3	75.2	20.0	19.0	104.1	11.0	19.9	107.1	14.0	25.0

TABLE 4.7: Execution times (T_i), idle-idle transition times (T_i^{I-I}) and average register usage (R_i) of processing cores in AMBA- and NoC-based decoders

As can be seen, AMBA-based decoder has similar register usage, R_i, as NoC for all four cores while decoding *test1.m2v* due to same processing cores between the two decoders (row 2, columns 3-6). However, as the registers in AMBA-based decoder are also used over idle period during bus arbitration, it has upto 7% lower register usage (given by (4.12)) than NoC-based decoder. Due shared-access of the bus among decoder cores, AMBA-based decoder has upto 2.18 times higher execution time for core MC compared to NoC-based decoder while decoding *test1.m2v*. Such time sharing of bus access also causes more idle-idle transition cycles in AMBA-based decoder, resulting in upto 6.9 times higher T_i^{I-I} compared to NoC-based decoder for core MC (row 2, column 6). With increased video sizes in other video bitstreams (*test2.m2v*, *test3.m2v* and *test4.m2v*), T_i and T_i^{I-I} increase but similar trend continues between AMBA- and NoC-based decoders for R_i, T_i and T_i^{I-I} values. Higher T_i results in higher number of SEUs experienced ($\mathcal{F}_{comp}$) in AMBA-based decoder compared to NoC-based decoder for decoding video different bitstreams (Table 4.1), as shown in Figure 4.15. The $\mathcal{F}_{comp}$ values are found using an arbitrary SER of 10^{-9} SEUs/bit/cycle in simulated fault injection environment (Section 4.5.1). The approximate $\mathcal{F}_{comp}$ values can also be validated through (4.11) with T_i, T_i^{I-I} and R_i values from Table 4.7. As expected, the AMBA-

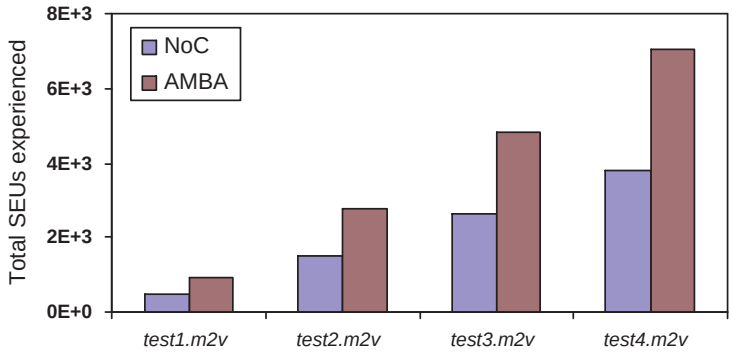


FIGURE 4.15: Comparative $\mathcal{F}_{comp}$ in AMBA- and NoC-based decoders for an arbitrary SER of 10^{-9}

based decoder experiences approximately 83% higher $\mathcal{F}_{comp}$ on average compared to NoC for decoding different video bitstreams. As a result of higher $\mathcal{F}_{comp}$, MPEG-2 decoder computation is expected to be affected more in AMBA-based decoder than NoC-based decoder. In Section 4.5.2.3 the impact of SEUs experienced is examined at application-level.

4.5.2.2 SEUs Experienced During Communication

An important aspect in the reliability of on-chip communication architectures is the number of SEUs experienced during inter-core data communication as these SEUs perturb the registers in the interconnects and affect the data transfer [143]. The number of

SEUs experienced during communication, $\mathcal{F}_{comm}$, depends on how the DTUs are transferred between communicating cores in an on-chip communication architecture. For a given SER (in SEUs per bit per cycle), the total $\mathcal{F}_{comm}$ of an on-chip communication architecture can be given by the product of per data transaction unit (DTU: packet for NoC and 32-bit data for AMBA) communication time, total number of DTUs transferred among cores, the register usage of the communication components and the SER. Hence $\mathcal{F}_{comm}$ can be expressed as

$$\mathcal{F}_{comm} = \sum_{j=1}^{M} \mathcal{N}_j L_{ch_j} R_{com_j} \lambda \quad , \tag{4.13}$$

where $\mathcal{M}$ is the number of inter-core communication links in the decoder ($\mathcal{M} = 4$, Figure 4.1(a)), $\mathcal{N}_j$ is the total number of DTUs between cores, L_{ch_j} is the channel latency (in clock cycles) and R_{com_j} is the average register usage in communication components during transfer of DTUs on j-th link. The channel latency, L_{ch_j} given by (4.5) gives a measure of communication time of DTUs within the on-chip communication architecture and is given by the time (in cycles) required for a DTU to be transferred from the output port of a processing core to an input port of target processing core. From Section 4.3.2.3, L_{ch} is 2 cycles for AMBA and 9 cycles for NoC with shortest path mapping between communicating cores (i.e. communicating cores are connected through $\mathcal{K}$=2 intermediate switches, Section 4.3.2.3). Note that L_{ch} for NoC varies for different floor mapping or packet routing algorithm. This is because floor mapping affects the number of intermediate switches travelled due to placement of cores on NoC tiles [49]. For example, L_{ch} increases to 15 and 20 clock cycles for floor mapping with 3 and 4 intermediate switches, respectively. Similarly, packet routing affects the channel latency since different communication paths result in varied number of intermediate switches travelled [41]. The average register usage of communication components during transfer of a DTU, R_{com_j} in (4.13), sets up another difference between AMBA- and NoC-based decoders. The R_{com_j} is given by

$$R_{com_j} = \frac{1}{(L_{ch_j} \mathcal{N}_j)} \sum_{n=1}^{\mathcal{N}_j} \sum_{l=1}^{L_{ch_j}} R_{n,l} \quad , \tag{4.14}$$

where $R_{n,l}$ is the instantaneous register usage on j-th link during inter-core communication of n-th DTU at l-th clock cycle (l=1:L_{ch_j}). For NoC-based decoder, $R_{n,l}$ in (4.14) includes registers used in packet overheads and buffers in NI interfaces, channels, virtual channels (VCs), and routers as packet is communicated between cores. For AMBA-based decoder, $R_{n,l}$ includes the registers used in address (HADDR), control signals (RD and WR), decoder and arbiter as DTU is communicated between cores. Using (4.14), R_{com_j} in NoC-based decoder (Figure 4.7) obtained from simulation logs is approximately 212 bits per data transfer cycle (using XY packet routing) and that in AMBA-based decoder

is approximately 87 bits per transfer cycle. The higher R_{com_j} of NoC is expected as NoC incorporates packet-based multihop routing with complex switch structure [41]. Note that R_{com_j} of NoC is dependent on the packet routing algorithm as underlying routing algorithm determines the switch design complexity and the associated the register usage [41]. For example, using source-based routing algorithm gives R_{com_j} value of 187 bits per cycle, while using odd-even routing algorithm results in R_{com_j} value of 273 bits per cycle as opposed to 212 bits per cycle for XY routing.

Video	$\mathcal{N}_1$ (VLD→MC), $\times 10^3$	$\mathcal{N}_2$ (VLD→ISQ), $\times 10^3$	$\mathcal{N}_3$ (ISQ→IDCT), $\times 10^3$	$\mathcal{N}_4$ (IDCT→MC), $\times 10^3$
test1.m2v	66	78	108	202
test2.m2v	232	273	364	666
test3.m2v	385	454	605	1111
test4.m2v	1598	1884	2503	4580

TABLE 4.8: Inter-core data transaction units (DTUs) of the MPEG-2 video decoder (Figure 4.1) for decoding different video bitstreams (Table 4.1)

Table 4.8 shows the number of DTUs, N_i (N_1 for VLD-MC link, N_2 for VLD-ISQ link, N_3 for ISQ-IDCT link, and N_4 for IDCT-MC link), recorded from simulation logs for MPEG-2 video decoder with four processing cores (Figure 4.1). Note that $\mathcal{N}$ values do not change between AMBA- and NoC-based decoders for a given video bitstream due to similar architecture for processing cores (Figure 4.1(a)). For decoding a given video bitstream, $\mathcal{N}$ is the least from core VLD to core ISQ. As the video decoding progresses with other cores, $\mathcal{N}$ between cores increases due to decompression of the original video bitstream. For example, only $\mathcal{N}$=66 $\times$ 10^3 DTUs are transferred from core VLD to core ISQ, while $\mathcal{N}$=202 $\times$ 10^3 DTUs are transferred from core IDCT to core MC for decoding *test1.m2v* (row 2, Table 4.8). For increased video sizes, $\mathcal{N}$ also increases for a given link. For example, 108 $\times$ 10^3 DTUs are transferred from core ISQ to core IDCT for decoding *test1.m2v* compared to 364 $\times$ 10^3 DTUs on the same link for decoding *test2.m2v* (column 4, Table 4.8). Figure 4.16 shows comparative $\mathcal{F}_{comm}$ of AMBA- and NoC-based decoders obtained from simulation logs for an arbitrary SER of 10^{-9}, while decoding different video bitstreams (Table 4.1). To demonstrate the impact of floor mapping, $\mathcal{F}_{comm}$ values of three different NoC configurations are shown with 2, 3 or 4 intermediate switches between cores. As expected, due to higher register usage (R_{com_j}) and channel latency (L_{ch}), NoC-based decoder links with 2 intermediate switches (Figure 4.7) suffer from 11 times higher $\mathcal{F}_{comm}$ compared to AMBA, which worsens to 18 and 24 times higher $\mathcal{F}_{comm}$ as number of intermediate switches increase to 3 and 4, while decoding *test1.m2v* (Figure 4.16). Similar trends between AMBA- and NoC-based decoders in terms of $\mathcal{F}_{comm}$ are also observed with other video bitstreams (Figure 4.16).

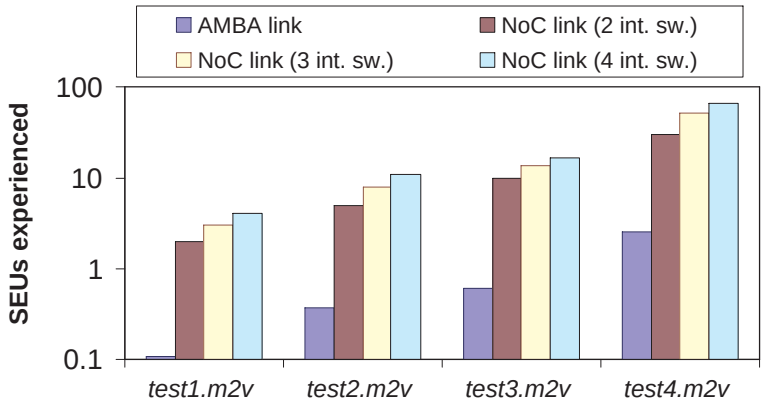


FIGURE 4.16: Comparative $\mathcal{F}_{comm}$ in interconnects of AMBA- and NoC- decoders for an arbitrary SER of 10^{-9}

To demonstrate the impact of choice of NoC packet routing algorithms on the $\mathcal{F}_{comm}$, Figure 4.17 shows the $\mathcal{F}_{comm}$ values for different packet routing algorithms: source-based, XY and odd-even routing algorithm implemented on NIRGAM [25]. The $\mathcal{F}_{comm}$ values are found with SER of 10^{-9}, while decoding the video bitstream *test4.m2v* (Table 4.1). The approximate values of $\mathcal{F}_{comm}$ can be found through (4.13) using the L_{ch} and R_{com_j} values of AMBA- and NoC-based decoders. As can be seen, using source-based

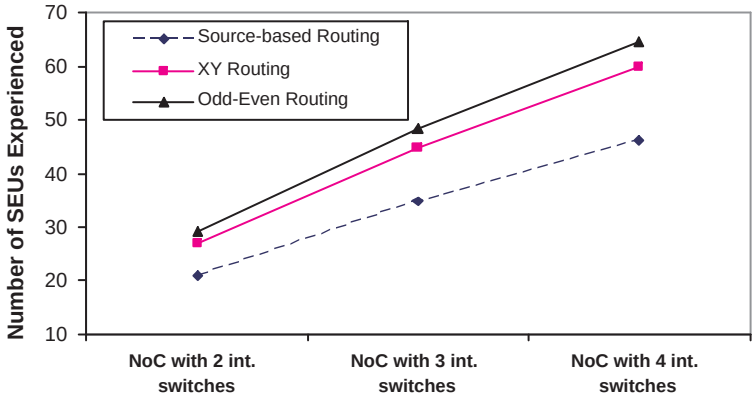


FIGURE 4.17: Impact of choice of routing algorithm on $\mathcal{F}_{comm}$ in NoC interconnects, while decoding *test4.m2v*

packet routing in NoC switches gives the least SEUs experienced during communication ($\mathcal{F}_{comm}$), while odd-even routing algorithm gives the highest $\mathcal{F}_{comm}$. This is because, due to source initiated routing information inserted in the packets, source-based routing gives the least register usage of 187 bits per cycle and simpler switch design. On the other hand, odd-even routing implements adaptive strategy of packet routing with a control mechanism to avoid deadlock and intermediate packet buffering, resulting in complex switch design and higher register usage of 273 bits per cycle. The XY routing has lower register usage (212 bits per cycle) than odd-even due to its deterministic

nature of choice of routing directions [144]. As expected, as more number of switches are travelled by NoC packets using these routing algorithms, the $\mathcal{F}_{comm}$ also increases linearly.

Comparing between $\mathcal{F}_{comm}$ (Figure 4.15) and $\mathcal{F}_{comp}$ values (Figure 4.16) of AMBA- and NoC-based decoders for decoding a given video bitstream, it can be seen that $\mathcal{F}_{comm} \ll \mathcal{F}_{comp}$. Nevertheless, $\mathcal{F}_{comm}$ affects the reliability on-chip communication as it leads to faults resulting in misrouting or loss of DTUs [143]. The loss of DTUs or misrouting causes the decoding process to be terminated or skip a number of video blocks or frames while decoding [145]. Next, the impact of overall SEUs experienced ($\mathcal{F}$) is evaluated at application-level.

4.5.2.3 Impact of SEUs at Application-Level

In Sections 4.5.2.1 and 4.5.2.2, the reliability of AMBA- and NoC-based decoders were investigated in terms of the SEUs experienced during computation ($\mathcal{F}_{comp}$) and communication ($\mathcal{F}_{comm}$). With the $\mathcal{F}_{comp}$ and $\mathcal{F}_{comm}$ values, the total number of SEUs experienced is given as

$$\begin{aligned} \mathcal{F} &= \mathcal{F}_{comp} + \mathcal{F}_{comm} \quad , \\ &= \left[\sum_{i=1}^{C} \left(T_i - T_i^{I-I}\right) R_i \lambda\right] + \left[\sum_{j=1}^{M} \mathcal{N}_j L_{ch_j} R_{com_j} \lambda\right] . \end{aligned} \tag{4.15}$$

In this section, the impact of injected SEUs, $\mathcal{F}$, given by (4.15), is evaluated at application-level. Such evaluation has also been used in [24] showing that the faults at architectural-level do not always lead to faults at application-level enabling low-cost fault tolerance mechanisms. The impact of $\mathcal{F}$ on decoder reliability has been evaluated using peak signal-to-noise ratio (PSNR) metric (as also used by [24]). PSNR is defined as

$$PSNR = 10 \log_{10} \frac{1}{PQ} \sum_{p=1}^{P} \sum_{q=1}^{Q} \frac{255^2}{(x_{p,q} - y_{p,q})^2} \quad (dB), \tag{4.16}$$

where P is the number of frames, each with Q pixels, $x_{p,q}$ and $y_{p,q}$ are the q-th pixels in p-th reference and decoded frames. Note that in the presence of SEUs, PSNR (given by (4.16)) is degraded due to alterations in computation registers containing $y_{p,q}$ values. As a result, the SEUs experienced during computation ($\mathcal{F}_{comp}$) have a direct impact on the PSNR. However, due to normalization with frames and pixels PSNR does not reflect temporal fidelity in the event of loss of frames [145]. To evaluate fidelity in the event of frame losses, frame error ratio (FER) metric has been used, which is defined as

$$FER = \frac{x}{P} \quad , \tag{4.17}$$

where x is the number of lost frames out of P frames. Frame losses during video decoding take place mostly due to misrouting of DTUs between communicating cores. Hence, SEUs experienced during communication ($\mathcal{F}_{comm}$) have a direct impact FER. The SEUs experienced during computation ($\mathcal{F}_{comp}$) has an indirect impact on FER as $\mathcal{F}_{comp}$ affects computation of video parameters and causes frame losses.

Figure 4.18(a) and (b) show the PSNR (in dB) and FER (in %) values of decoded video frames found through (4.16) and (4.17), while decoding video bitstream *test4.m2v* in AMBA- and NoC-based decoders. The PSNR and FER values of NoC-based decoder are observed for three different NoC configurations: with 2, 3 and 4 intermediate switches between communicating cores. An arbitrary SER of 10^{-9} SEUs per bit per cycle is used

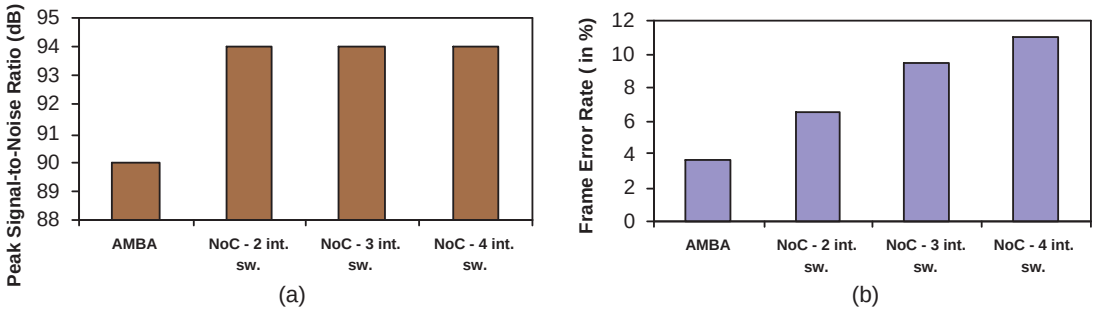


FIGURE 4.18: (a) Comparative PSNRs of AMBA- and NoC-based decoders, while decoding *test4.m2v*, (b) comparative FERs of AMBA- (Figure 4.3) and NoC-based decoders (Figure 4.7), while decoding *test4.m2v*

in simulated fault injection environment (Section 4.5.1). As expected, NoC-based decoder outperforms AMBA-based decoder with upto 4dB higher PSNR (Figure 4.18(a)). This is because NoC-based decoder experiences lower $\mathcal{F}_{comp}$ than AMBA-based decoder (Section 4.5.2.1). However, since PSNR reflects the fidelity of video blocks due to perturbation of registers by $\mathcal{F}_{comp}$, NoC-based decoder shows similar PSNRs for all configurations (the number of intermediate switches does not affect $\mathcal{F}_{comp}$, Section 4.5.2.2). Comparing the FER values in Figure 4.18, it can be seen that AMBA-based decoder gives 3% lower FER compared to NoC-based decoder configuration with 2 intermediate switches due to higher number of SEUs experienced during communication, $\mathcal{F}_{comm}$ (Section 4.5.2.2). As expected, with increased number of intermediate switches, NoC-based decoder experiences higher FER due to increased $\mathcal{F}_{comm}$ given by (4.13). For example, FER of NoC-based decoder increases from 6.5% to 9.5% and 11% as number of intermediate switches increases from 2 to 3 and 4 (Figure 4.18).

The FER values of NoC-based decoder in Figure 4.18(b) are obtained using XY packet routing algorithm. Figure 4.19 demonstrates the impact of choice of routing algorithm on the FER of the NoC-based decoder (Figure 4.7), while decoding the video bitstream *test4.m2v* (Table 4.1). Three different packet routing algorithms are used: source-based, XY and odd-even. FER values are obtained through (4.17) from decoded video frames in

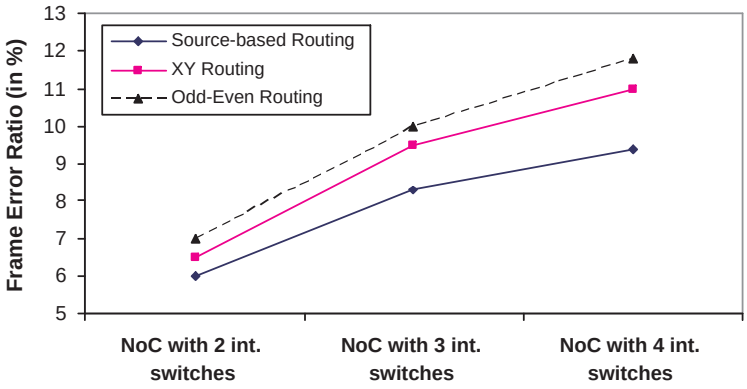


FIGURE 4.19: Impact of choice of routing algorithm on the FER of NoC-based decoder, while decoding *test4.m2v*

SystemC fault injection environment with an SER of 10^{-9} (Section 4.5.1). As expected, using the source-based packet routing algorithm gives the lowest FER among the routing algorithms due to the lowest $\mathcal{F}_{comm}$ (Section 4.5.2.2). The NoC-based decoder employing XY and odd-even routing algorithm give higher FER due to the higher $\mathcal{F}_{comm}$. It can be seen that with increasing number of intermediate switches between communicating cores, the FER of the NoC-based decoder increases almost linearly due to increased $\mathcal{F}_{comm}$, given by (4.13).

4.5.3 Impact of Architecture Allocation

To demonstrate the impact of architecture allocation on reliability, the different number of allocated cores were simulated using the task mapping in Table 4.5. Table 4.9 shows the number of SEUs experienced during computation and communication in AMBA- and NoC-based decoders for architecture allocations of 2 cores, 3 cores and 5 cores (simulation results for architecture allocation of 4 cores are shown in Tables 4.7 and 4.8).

Video	Arch.	2 Cores		3 Cores		5 Cores	
		$\mathcal{F}_{comp}$	$\mathcal{F}_{comm}$	$\mathcal{F}_{comp}$	$\mathcal{F}_{comm}$	$\mathcal{F}_{comp}$	$\mathcal{F}_{comm}$
test1.m2v	NoC	3.94E+2	1	4.44E+2	1	5.45E+2	1
	AMBA	7.02E+2	0	7.97E+2	0	9.78E+2	0
test2.m2v	NoC	1.20E+3	3	1.36E+3	4	1.66E+3	4
	AMBA	2.16E+3	0	2.43E+3	0	2.93E+3	1
test3.m2v	NoC	2.03E+3	5	2.34E+3	6	2.83E+3	8
	AMBA	3.65E+3	1	4.28E+3	2	5.19E+3	3
test4.m2v	NoC	3.06E+4	22	3.48E+3	25	4.21E+3	31
	AMBA	5.51E+3	2	6.24E+3	3	7.44E+3	5

TABLE 4.9: Impact of architecture allocation on the reliability in terms of number of SEUs experienced during computation ($\mathcal{F}_{comp}$) and communication ($\mathcal{F}_{comm}$)

The number of SEUs experienced during computation ($\mathcal{F}_{comp}$) and the number of SEUs experienced during communication ($\mathcal{F}_{comm}$) of architecture with 2 allocated cores is shown in column 3, while that of 3 and 5 cores are shown in columns 4 and 5 (Table 4.9). As expected, NoC-based decoder experiences less number of SEUs during computation than AMBA, while AMBA experiences less SEUs during communication for all architecture allocations. It can be seen that both AMBA- and NoC-based decoders experience higher $\mathcal{F}_{comp}$ as the number of allocated cores increases in the architectures. This is because with higher number of allocated cores, the overall register usage ($R = \sum_i R_i$) increases due to duplication of shared task registers among cores, which eventually results in higher $\mathcal{F}_{comp}$, given by (4.11). Also, the $\mathcal{F}_{comm}$ for architectures with higher number of allocated cores increases due to increase in the number of links and number of DTUs being communicated among processing cores.

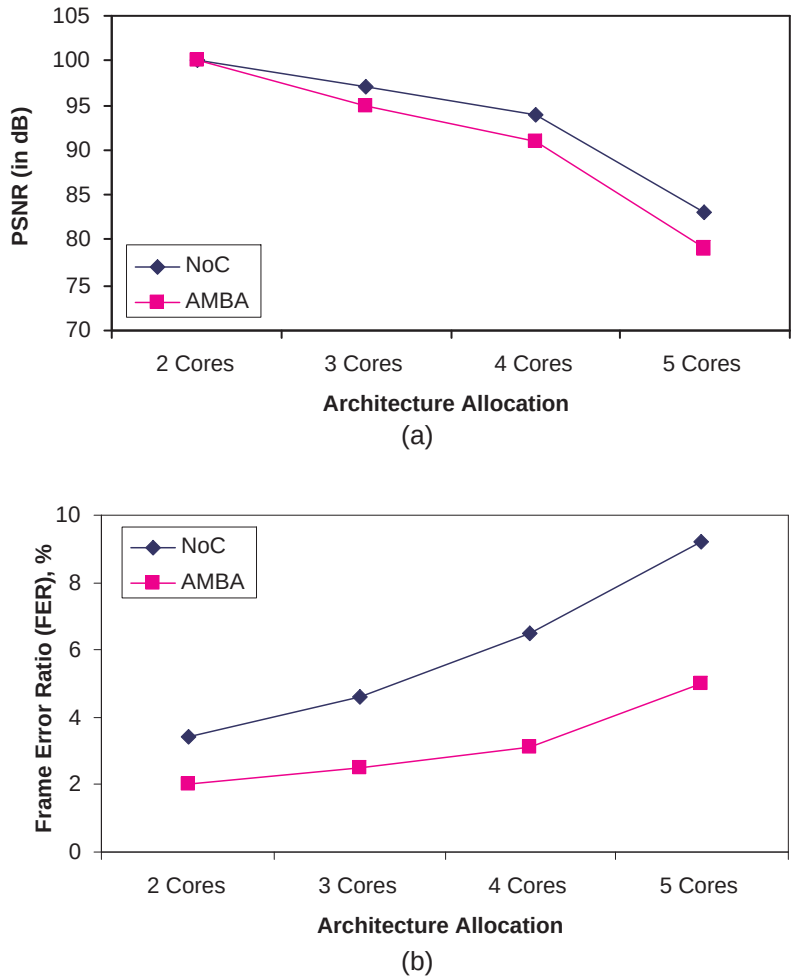


FIGURE 4.20: (a) Comparative PSNRs of AMBA- and NoC-based decoders for different architecture allocations while decoding *test4.m2v*, (b) Comparative FERs of AMBA- and NoC-based decoders for different architecture allocations, while decoding *test4.m2v*

To observe the impact of total number of SEUs experienced at application-level, Figure 4.20 shows the corresponding PSNR and FER values of the different allocated cores

of AMBA- and NoC-based decoders for decoding *test4.m2v*. The PSNR and FER values were found at SER of 10^{-9} using the video bitstream *test4.m2v*. As can be seen, NoC-based decoder gives better PSNR for all architecture allocations (Figure 4.20(a)) due to lower number of SEUs experienced during computation, $\mathcal{F}_{comp}$ (Section 4.5.2.1). Due to increased $\mathcal{F}_{comp}$ for increasing number of allocated cores, architecture with higher number of cores give poorer PSNRs for AMBA- and NoC-based decoders. For example, PSNR decreases from 99dB for architecture with 2 cores to 84dB for architecture with 5 cores for AMBA-based decoder (Figure 4.20(a)). As expected, decoder architecture with higher number of cores gives higher FER due to increased $\mathcal{F}_{comm}$ (Table 4.6). For example, FER increases from 2% for architecture with 2 cores to 4.5% for architecture with 5 cores for AMBA-based decoder (Figure 4.20(b)). Note that AMBA-based decoder gives lower FER when compared with NoC-based decoder due to lower number of SEUs experienced during communication, $\mathcal{F}_{comm}$ (Section 4.5.2.2).

4.6 Concluding Remarks

This chapter has presented a comparative analysis between shared-bus AMBA and NoC in terms of performance and reliability using real application traffic of MPEG-2 video decoder in cycle-accurate simulation environment. Supported by analytical and simulation results, it has been shown that the NoC-based decoder reduces decoding time by a factor of 2.18 on average compared to AMBA-based decoder (Section 4.3) for the MPSoC architecture with four processing cores (Figure 4.1). Despite higher channel latency, NoC-based decoder has higher core concurrency, core efficiency and bandwidth advantage over AMBA-based decoder and can operate at lower frequency (approximately 43%) than AMBA-based decoder (Section 4.4). It has also been shown that AMBA-based decoder experiences higher SEUs for a given soft error rate during computational processing due to higher execution time and NoC-based decoder experiences higher SEUs during packet communication due to higher register usage and channel latency (Section 4.5). Considering reliability at application-level in terms of peak signal-to-noise ratio (PSNR), it was shown that NoC-based decoder has upto 4dB higher PSNR compared to AMBA-based decoder. However, due to higher communication SEUs injected, NoC-based decoder gives upto 3% higher MPEG frame error ratio (FER) . Furthermore, the impact of architecture allocation on the performance and reliability were shown. The comparisons in this chapter have focused on performance and reliability aspects between NoC and AMBA, while it supports the previously reported comparisons involving power, area and scalability in [6, 79].

Chapter 5

Voltage Scaling Technique for Power Minimisation

Chapter 4 presented a comparative analysis of performance and reliability between two on-chip communication architectures: shared-bus AMBA and network-on-chip (NoC). The comparisons were carried out with the aim to identify an efficient on-chip communication architecture for multiprocessor system-on-chip (MPSoC). Power minimisation is a key design objective of MPSoCs used in hand-held devices to extend battery life [89] and is one of the main aims of this chapter. To address the power minimisation issue, dynamic voltage scaling (DVS) technique has recently been proposed [55, 146] (a brief introduction to DVS technique is presented in Section 2.2, Chapter 2). The DVS technique reduces the dynamic power consumption by decreasing the operating frequency and supply voltage of a processing core in an MPSoC depending on its workload [147]. However, lowering the supply voltage reduces the critical charge of a circuit node required to induce a soft error by particle hit, causing exponential increase of soft errors and degradation of reliability [88, 148] (a brief introduction to reliability is presented in Section 2.3, Chapter 2). As a result, there is a trade-off between power minimisation and reliability, which has been investigated recently in a number of publications, such as [63, 91, 88, 148, 149].

At present there is growing interest in evaluating the impact of soft errors at application-level rather than architectural-level, particularly in multimedia applications, to optimise system design. This has recently led to the concept of application-level correctness [23]. In this chapter, a relationship between application-level correctness and system-level power management using voltage scaling technique is established. Based on this relationship, a novel power minimisation technique is proposed to generate designs that are optimised in terms of power consumption, while providing acceptable application-level correctness and meeting a specified real-time performance deadline. The effectiveness of the proposed technique is evaluated using an MPEG-2 video decoder as a case study

and with peak signal-to-noise ratio (PSNR) as the application-level correctness metric. Furthermore, using the proposed voltage scaling technique the impact of varying the number of processing cores (architecture allocation) and application task mapping (distribution of tasks among cores of the MPSoC architecture) is investigated on the trade-offs between power minimisation and application-level correctness.

The rest of the chapter is organised as follows. Section 5.1 presents a review of related works and Section 5.2 introduces the preliminaries. Section 5.3 outlines the motivation of the proposed work and Section 5.4 establishes the relationship between application-level correctness and power consumption leading to the development of the proposed voltage scaling technique. Section 5.5 presents the experimental results and Section 5.6 investigates the impact of application task mapping (distribution of application tasks among cores) and architecture allocation (varying the number of processing cores) on the trade-off between application-level correctness and power minimisation. Section 5.7 provides further validation of the proposed power minimisation technique. Finally, Section 5.8 concludes the chapter.

5.1 Related Works

Over the last decade, a number of techniques have been proposed to mitigate the effect of soft errors in low power system design. Examples of such techniques used are redundancy [89, 123], slack time scheduling, application check-pointing [63, 96], mapping and scheduling of fault tolerance policies [74, 99] (a brief review of the different fault tolerance techniques is presented in Section 6.1, Chapter 6). The fault tolerance achieved through these techniques are evaluated by different architectural- or application-level metrics. A brief account of recently reported metrics follows.

Architectural-level evaluation of fault tolerance is described by a number of different metrics. The fault tolerance cost of these metrics is a direct implication of the number of faults seen at the architectural-level [150]. As a result, achieving high fault tolerance can be expensive in terms of resources and power consumption using such metrics [61]. A commonly used architectural-level metric is the number of faults tolerated for a given number of faults injected. Hardware architecture solutions proposed in [151, 152], check-pointing-based fault tolerance technique proposed in [96, 97] and task replication and re-execution-based fault tolerance technique proposed in [95] use this metric to tolerate a given number of faults. The number of faults that can be tolerated in these techniques is limited by the overhead of hardware or system resources that is incurred.

Probabilistic fault rate is also an effective architectural-level metric to describe the presence of soft errors [153]. Different parameters have been proposed to date describing probabilistic fault rates, such as mean-time-to-failure (MTTF), fault injection time

(FIT), soft error rate (SER) and fault injection rate (FIR) (a brief introduction to different probabilistic fault rates used in fault injection is presented in Section 2.3, Chapter 2). The effectiveness of a fault tolerance technique using these metrics is demonstrated by the improvement of the underlying fault rate as opposed to tolerating every fault present in the system. As a result, the fault tolerance cost is greatly relaxed using probabilistic fault rate metric. A number of fault tolerance techniques have been reported over the years using this metric showing associated costs and trade-offs. For example, a convex power minimisation problem has been presented using the relationship between soft error rate (SER) and the dynamic voltage scaling (DVS) in [14]. Similarly, in [9] a power minimisation approach has been presented using the relationship between SER and gate sizes. Another approach using an SER-based characterisation of fault tolerance is used in [62] showing the trade-off between combined time and information redundancy and power minimisation through DVS. In [74] fault tolerance is evaluated as improved fault duration, which is achieved through mapping of fault tolerance policies to different processes for real-time systems. Among other techniques reported, a power constrained redundancy-based fault tolerance technique using MTTF is reported in [89] showing the relationship between achievable fault tolerance and additional costs required. Also, a soft error hardening technique is reported in [154] showing overhead in terms of power-delay-area product.

Another effective architectural-level metric is to evaluate fault tolerance using a reliability metric [100]. Reliability metric of an architectural component is expressed as a function of fault rate and the associated costs of different parameters that affect reliability of the component [155]. The overall reliability is expressed as a function of the component reliabilities with appropriate weights to them depending on their impact on the overall reliability. A reliability-centric high-level synthesis technique for embedded systems is proposed in [156]. It is shown that the increased reliability causes higher latency in the system and higher area due to additional resources required. A number of studies have also demonstrated the trade-off between achievable reliability and resource overheads, e.g. in terms of hardware resources [157], latency [158] and scheduling length [99].

Recently, different techniques have been proposed using application-level metrics to ease fault tolerance requirements in the presence of errors. The aim is to evaluate the impact of errors at application-level rather than architectural-level to reduce system cost. The reduction of cost using these metrics depends on the design and application being considered [159]. For example, according to [160, 161, 162] between 10%-40% of soft errors in flip-flips result in data corruption in the memory at the application-level. This means that between 60%-90% of soft errors do not have any effect on the system at application-level. In another study of the effect of faults on applications, [7, 8] showed that up to 0.2% error density is acceptable in multimedia applications. A similar finding has recently been reported in [23, 24] showing that the soft errors at the architectural-

level do not always lead to the errors at the application level . For example, using an MPEG-2 video decoder it was shown that upto 46% of the single-event upsets injected at application registers and queues do not have any impact at the application-level. The ability of applications to mask the micro-architectural faults has been used to define the concept of application-level correctness in [23, 24]. Because of the relaxed requirements of application-level correctness, the authors of [23, 24] showed that application-level correctness can be exploited for low-cost fault tolerance of embedded systems.

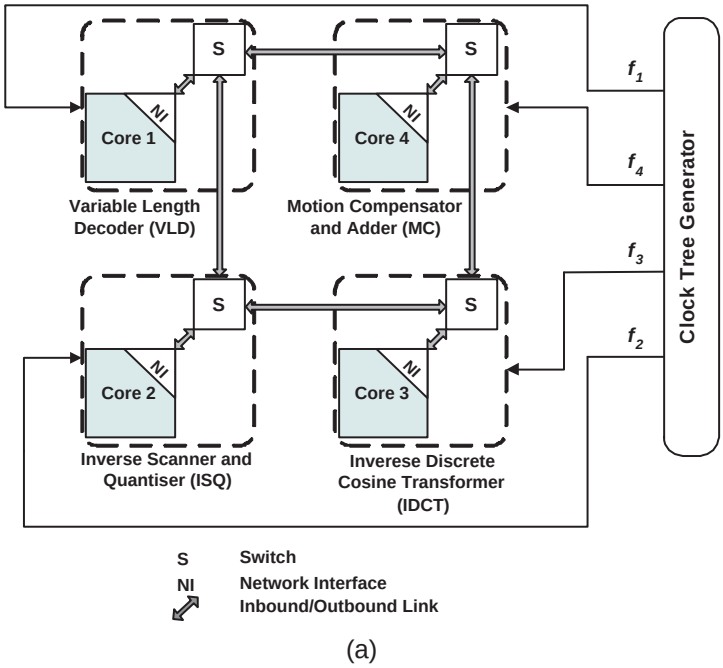
To exploit the advantage of low-cost fault tolerance using application-level correctness for power minimisation in DVS-enabled systems, the relationship between application-level correctness and voltage scaling is essential. Currently, no such relationship exists. In this chapter, a relationship between application-level correctness and voltage scaling is developed leading to a novel voltage scaling technique for system-level power minimisation, while meeting real-time constraint and maintaining acceptable application-level correctness.

5.2 Preliminaries

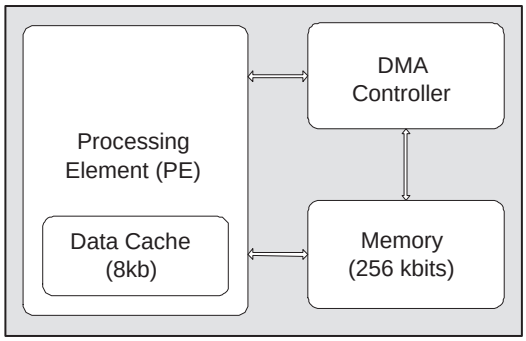
In this section, the application model, system architecture and fault injection model are introduced and the concept of application-level correctness is briefly explained.

5.2.1 Application and Architecture

MPEG-2 video decoder constitutes a major component of the current and future multimedia systems (refer to Appendix A for further details regarding MPEG-2 video decoder). In this work, the MPEG-2 video decoder has been chosen as a case study (applicability to other applications are explained in Section 5.7). Due to high performance and reliability (Sections 4.4 and 4.5, Chapter 4), two-dimensional mesh-based NoC is chosen as the multiprocessor system-on-chip (MPSoC) architecture. Figure 5.1(a) shows an MPEG-2 video decoder using a mesh-based NoC architecture with four processing cores. The architecture allocation (choice of number of processing cores) and application task mapping (distribution of application tasks among cores) are carried out arbitrarily to reflect MPSoC. The impact of architecture allocation and application task mapping is investigated in Section 5.6. Figure 5.1(b) shows the block diagram of a processing core in NoC. Each core has 8k bytes of data cache, 8k bytes of instruction cache and 256k bytes of local memory accessed through direct memory access (DMA) controller. The cache and memory sizes are chosen to provide high availability of data and parallelism among the processing cores. A clock tree generator is included to provide support for voltage scaling in the MPSoC (details of the voltage scaling arrangement is presented in Section 5.2.2). Multiprocessor ARM (MPARM), a cycle-accurate simulator



(a)



(b)

FIGURE 5.1: (a) An MPEG-2 video decoder MPSoC architecture based on two-dimensional mesh-based NoC using four processing cores with support for voltage scaling, and (b) block diagram of a processing core within the NoC tile used in this work

for ARM7TDMI processor proposed in [21], is used to obtain the power consumption reported in this work. MPARM is based on real-time executive for embedded systems (RTEMS [163]), which gives hardware-software (HW/SW) co-design and simulation environment with software implementation (such as MPEG-2 video decoder) using C/C++ and hardware implementation (such as processing elements and interconnects) using SystemC. Employing cycle-accurate simulations, MPARM can profile system performance and power estimation of the MPSoC application. For more details regarding MPARM, see Section B.2, Appendix B.

MPEG-2 decoder operates with video frames of various resolutions and rates. In this work, three video sequences of different frame resolutions and decoding rates are used for

experimental purposes (Section 5.5) as shown in Table 5.1 (for different frame formats and decoding rates used in MPEG-2 decoder, see Section A.1.1, Appendix A).

Sequence	**Frames**	**Resolution, pixels**	**Decoding rate** *(frames/sec)*
house†	623	QCIF:176 × 144	25 (PAL)
tennis‡	437	CIF:352 × 240	29 (NTSC)
flower‡	440	2CIF:352 × 576	25 (PAL)

†Source: http://www.cortonaweb.net/eng/video/

‡Source: ftp://ftp.tek.com/tv/test/streams/Element/

TABLE 5.1: Test video sequences used for experimental purposes (Section 5.5)

5.2.2 Voltage Scaling

Voltage Scaling is an effective technique for power minimisation [48, 149] (see Section 2.2, Chapter 2 for further details related to voltage scaling). To introduce power minimisation in the MPSoC decoder through the voltage scaling technique, a clock tree generator is included (Figure 5.1(a)). Using this clock tree generator different processing cores within the MPSoC are provided with different clock frequencies and operating voltages. Figure 5.2(a) shows the voltage scaling arrangements for the processing core and switch. As can be seen, two different clock frequencies are fed at the two ends of the NI: nominal clock frequency (the maximum operating frequency, with no scaling) is attached to the switch end and the clock with scaling is attached to the processing core end (Figure 5.2(a)). The maximum available clock frequency is attached to the switch end to keep the communication latency as low as possible [164]. Figure 5.2(b) shows the voltage scaling arrangements for the network interface (NI) within the NoC tile, which acts as an interface between switch and processing core. As can be seen, the NI plays an important role in synchronising the two different clock frequencies: the first one (with no scaling) drives the NI front-end, the side to which the switch is attached, and the second one (with scaling through a clock divider) drives the NI back-end, the side to which the processing core is attached (Figure 5.2(b)). The first FIFO buffers within the NI incorporate necessary interface to enable synchronised communication between the two ends. This process introduces further delay in channel latency, increasing the original channel latency by 2 clock cycles for lower scaling of clock frequency by a factor of 1. For example, the channel latency can increase from 9 clock cycles to 11, 13 and 15 clock cycles for scaling the operating clock frequency from nominal clock frequency (200MHz, at the switch end) to the operating clock frequencies 100MHz, 66.7MHz and 50MHz (at the processing core end). Due to pipelined communication, increased channel latency does not affect the inter-core communication greatly. Similar voltage scaling

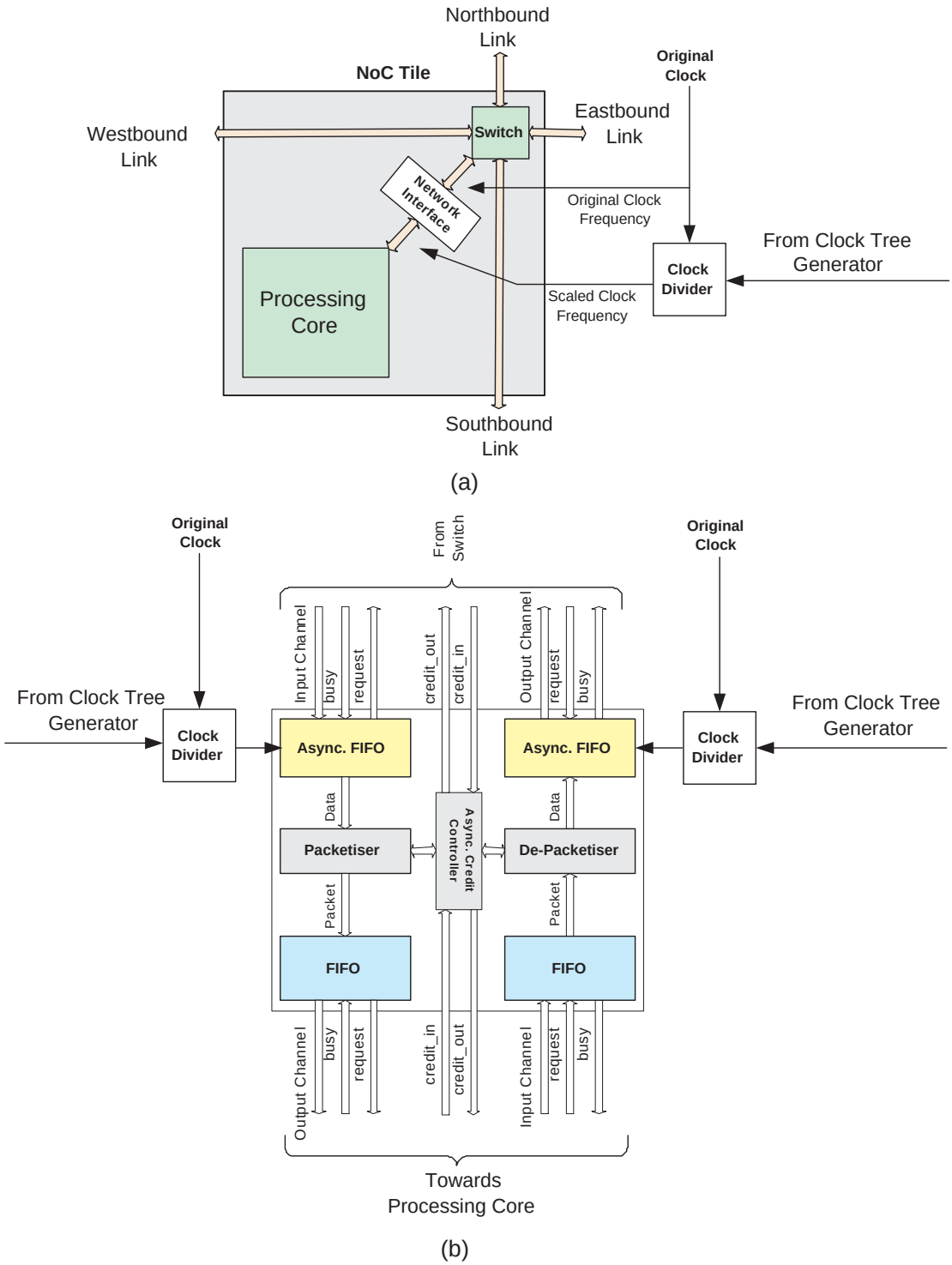


FIGURE 5.2: (a) Block diagram of the voltage scaling arrangements used in the NoC tile, and (b) block diagram of the NoC network interface (NI) with asynchronous FIFO buffer and interface between different clock frequencies to enable synchronised communication between the NI ends

arrangement (Figure 5.2(a) and (b)) is also used in [164].

For a given voltage scaling, the dynamic power consumption, P, of a processing core is given as,

$$P = \alpha C_L f {V_{dd}}^2 = \alpha P_{max} \tag{5.1}$$

where C_L is the processor load capacitance per cycle, V_{dd} is the supply voltage, f is the operating frequency, α is the processor activity factor and P_{max} is the maximum dynamic power ($P_{max} = C_L f {V_{dd}}^2$). While C_L, f and V_{dd} are dependent on process technology, α is a fraction between 0 and 1 indicating how often clock ticks lead to switching activity on average and depends on the application being executed on the processor [165]. The DVS technique effectively reduces power consumption in (5.1) by scaling down V_{dd} and f. For ARM7TDMI processor, the relationship between V_{dd} (in volts) and f (in MHz) is given by [47] as,

$$V_{dd}(f, s) = \left[0.1667 + \frac{4.1667 \times f}{10^3 \times s}\right] \tag{5.2}$$

where s is the frequency scaling constant. According to [47], ARM7TDMI has nominal supply voltage of V_{dd}=1V operating at f=200MHz and voltage scaling gradually stalls below f=90MHz due to (5.2). Using (5.2), the available voltage-frequency scaling options (eg. with scaling factor s=1, 2 and 3 in (5.2)) are shown in Table 5.2.

Scaling, s	f, *MHz*	V_{dd}, *V*
1	200	1
2	100	0.58
3	66.67	0.44

TABLE 5.2: Operating frequency, f and supply voltage, V_{dd}, for different scaling options for ARM7TDMI processor

5.2.3 Fault Injection Model

In this work, SEU-based fault injection is carried out using the fault injection simulator proposed in Chapter 3. The injection of SEUs using this simulator is initiated through replacement of variable or signal types in the original design specification to equivalent fault injection enabler types. Such type replacement enables the formation of a fault locations database, which contains the target registers for SEU injection. The fault injection simulator injects SEUs in these target registers based on the specified soft error rates and probability distribution for determining fault locations. Figure 5.3 shows the fault injection setup used for the MPEG-2 decoder with four processing cores (Figure 5.1). As can be seen, due to type replacement to fault injection enabler types four fault locations databases are formed for the four processing cores (for example, fault locations database 1 for core VLD, fault locations database 2 for core ISQ and so on).

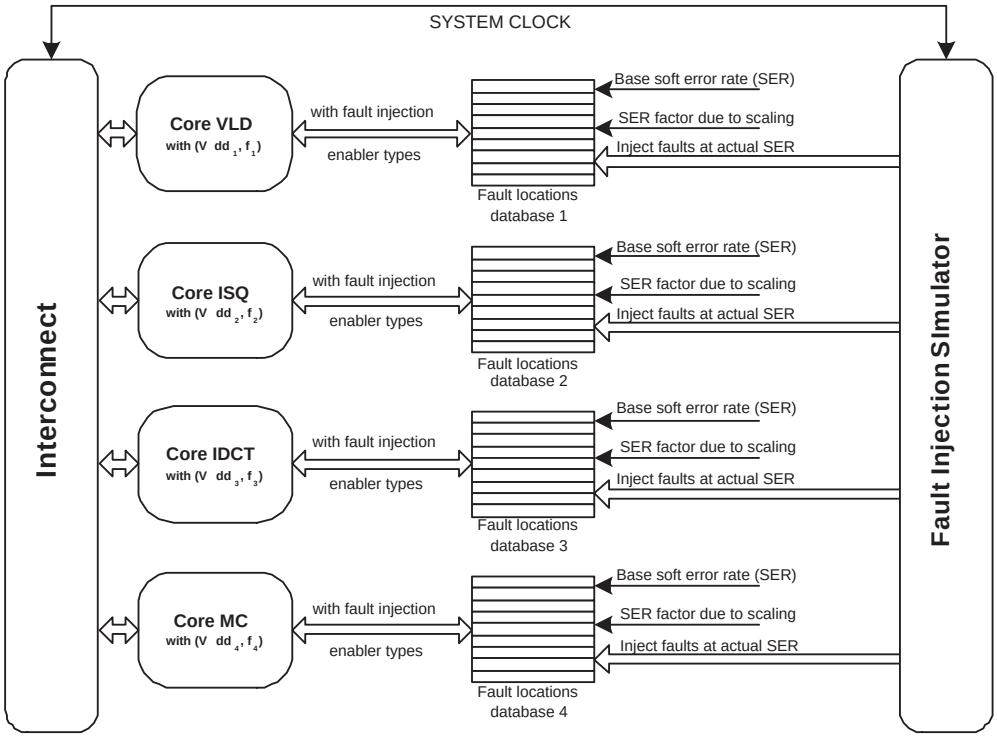


FIGURE 5.3: Fault injection setup for MPSoC decoder with four processing cores (Figure 5.1)

For a given base soft error rate (SER, λ_0, in number of SEUs per bit per cycle considering no scaling), the actual SERs (λ, after scaling) for processing cores are found by the corresponding voltage settings used (Figure 5.3). For the resulting SERs, the SEUs are injected at random locations determined by Poisson's distribution in the target registers of the fault locations databases. To control the fault injection timing, the system clock of the MPEG-2 video decoder is connected to the fault injection simulator (Figure 5.3).

5.2.4 Application-level Correctness

Application-level correctness is a measure of the impact of architectural faults or soft errors at application-level. It was introduced in [24, 166] showing that the faults at architectural-level do not always lead to faults at application level, leading to low-cost fault tolerance techniques. Depending on the nature of outcome and computation carried out, the application-level correctness metric can be different for various applications. For example, peak signal-to-noise ratio (PSNR) was used as application-level correctness metric for MPEG-2 video decoder in [24]. In this work, PSNR is also used to determine the objective quality of MPEG-2 video in the presence of SEUs and the corresponding application-level correctness. The application-level correctness in terms of PSNR, denoted by Ω, defines the the ratio of peak video signal power to the mean-squared-error

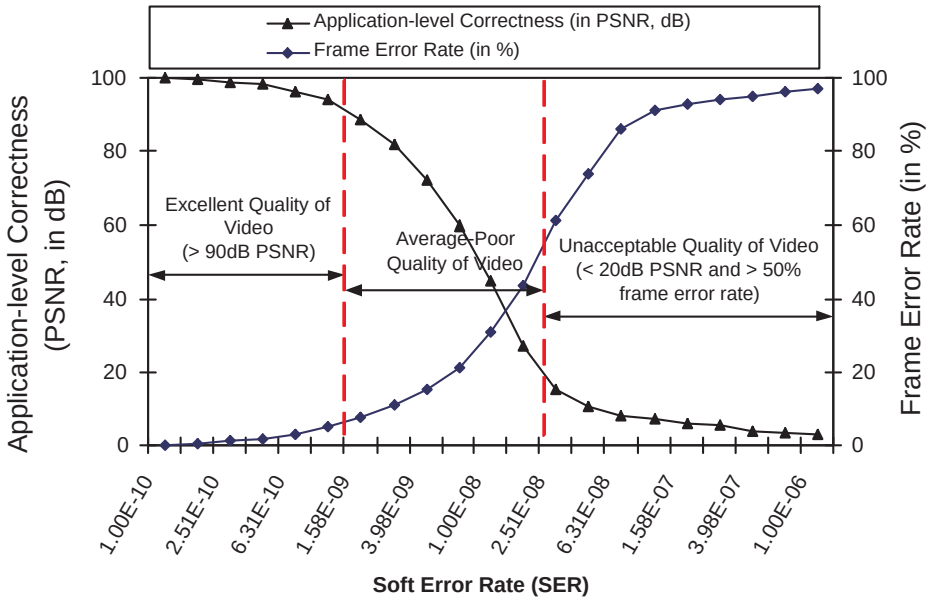


FIGURE 5.4: Application-level correctness in terms of PSNR (in dB) and frame error rate (in %) of decoded frames of *tennis* video sequence (Table 5.1)

or noise power as

$$\Omega = PSNR = 10\log_{10}\frac{1}{ML}\sum_{m=1}^{M}\sum_{l=1}^{L}\frac{255^2}{(x_{m,l}-y_{m,l})^2} \quad (dB), \tag{5.3}$$

where M is the number of frames in the video sequence, L is the number of pixels in a frame, $x_{m,l}$ and $y_{m,l}$ are the l-th pixels in m-th reference and decoded frames. Note that in the presence of SEUs, PSNR (given by (5.3)) is degraded due to alterations in the registers that contain $y_{m,l}$ values. Other registers related to header sequence and application flow control are also perturbed by these SEUs and causes frame errors or losses. As a result, degradation of PSNR (in dB) is related to amount of frame losses in the presence of SEUs [23]. Figure 5.4 shows the relationship between PSNR (in dB) and frame error rate (FER, in %) for varying soft error rate (SER) from 10^{-10} to 10^{-6} using the *tennis* video sequence (Table 5.1). As expected, with increasing SER, PSNR (in dB) is degraded and FER (in %) is increased due to increased number of SEUs experienced by the decoder (the variation of PSNR over increasing SERs is further explained in Section 5.4.3). At low SERs, PSNR equal to or more than $90dB$ corresponds to excellent quality of video with negligible frame loss (about 5% or below) [23] (bounded region on left). With increased SERs, PSNR between $90dB$ and $20dB$ give average to poor quality of video with less than 50% FER [24] (bounded region in the middle). As the SER increases further, PSNR reduces to $20dB$ or below, corresponding to high FER of more than 50% (bounded region on right, Figure 5.4). Such high FER and low PSNR gives an unacceptable quality for real-time video [167]. In [168, 169], PSNR of $30dB$ has been

considered as an acceptable application-level correctness (Ω) for video applications. In this work, $30dB$ PSNR is also considered as the acceptable application-level correctness (Ω).

5.3 Motivation

Power reduction through voltage scaling increases the soft error rate (SER) exponentially [148]. The SER due to voltage scaling, λ, is expressed as a function of the base SER (i.e. the SER with no voltage scaling), λ_0, as [62]

$$\lambda = \lambda_0 . 10^{\frac{V_{max} - V_{dd}}{S}}, \tag{5.4}$$

where V_{max} is the maximum supply voltage (with no scaling), V_{dd} is the supply voltage with scaling and S is the voltage value such that when supply voltage decreases by S, the SER increases by one order of magnitude. According to [170], for 90-nm CMOS technology the SER increases by 18 percent when V_{dd} is reduced by 10 percent. This gives an S value of 1.39 in (5.4). Using this S value in (5.4), the SER increases by approximately 2.5 and 3.3 times when V_{dd} is scaled from 1V to 0.58V and to 0.44V, respectively. Such increase in SER leads to higher number of soft errors and reduction in the PSNR (as described in Section 5.2.4). To demonstrate the impact of voltage scaling on application-level correctness, Table 5.3 shows the PSNR values (Ω, in dB) found using (5.3) and power values found through MPARM [21] assuming a base SER of 3.98×10^{-8} SEUs per bit per clock cycle (equivalent to 1 fault per bit in every 250ms) for the MPEG-2 decoder cores (Figure 5.1(a)). Depending on the technology feature size and operating conditions, SER can vary and therefore, different values has been reported [65]. For example, SER of 10^{-7} SEUs per bit per clock cycle was reported for memory in [171] and SER of 10^{-2} SEUs/second (corresponding to SER of 10^{-10} SEUs per bit per clock cycle for 100MHz memory speed) was reported in [91]. The different scaling options are shown in column 1, while power consumption (in mW) and PSNR values (Ω, in dB) for each these scaling options are shown in columns 2 and 3 (Table 5.3). The PSNR values (column 3, Table 5.3) are found through decoding the *tennis* video sequence (Table 5.1) and are evaluated using SystemC fault injection experiments (Section 5.2.3).

Scaling, s	**Power** (P), mW	**PSNR** (Ω), dB
1	27.2	63 ($\lambda = 3.98 \times 10^{-8}$)
2	4.9	33 ($\lambda = 9.65 \times 10^{-8}$)
3	2.5	23 ($\lambda = 1.21 \times 10^{-7}$)

TABLE 5.3: Power consumption (in mW) and PSNRs (in dB) for different voltage scaling options at $\lambda = 3.98 \times 10^{-8}$

From Table 5.3, it can be seen that the application-level correctness (Ω) in terms of PSNR is the best ($63dB$) when no voltage scaling is applied (i.e. s=1, all processing cores of the decoder operate with V_{dd}=1V and f=200MHz). However, this is achieved at the cost of the highest power consumption of 27.2mW (column 2, Table 5.3). As lower voltage scaling is applied (s=2 or s=3) to reduce the power consumption, the power minimisation is achieved at the cost of reduced PSNR. For example, considering the case when s=2 (i.e. all four processing cores of the decoder, Figure 5.1(a), operate with V_{dd}=0.58V and f=100MHz), the PSNR is reduced to $33dB$ compared to $63dB$ for the case when no voltage scaling is used (i.e. s=1, all cores operate with V_{dd}=1V and f=200MHz). This is due to the increase in the SER from 3.98×10^{-8} to 9.65×10^{-8} given by (5.4) arising out of voltage scaling of s=2 as shown in column 3, Table 5.3. Note that it is possible to reduce the power consumption further by using s=3 (i.e. all four processing cores of the decoder of Figure 5.1(a) operate with V_{dd}=0.44V and f=66.7MHz). However, using such voltage scaling causes an increase in the SER from 3.98×10^{-8} to 1.21×10^{-7} (column 3, Table 5.3) given by (5.4). This leads to an unacceptable level of application-level correctness (Ω) of $23dB$ PSNR, since the minimum acceptable application-level correctness (Ω_{ref}) assumed in this work is $30dB$ PSNR.

Table 5.3 shows that power reduction without consideration of application-level correctness leads to unacceptable application-level correctness (as in the case of s=3). This poses an interesting question as to how the voltage scaling should be chosen for the decoder processing cores to achieve the minimum power, while the application-level correctness and real-time performance are maintained at acceptable levels.

5.4 Proposed Voltage Scaling Technique

To answer this question, the relationship between application-level correctness and power minimisation through voltage scaling of the MPSoC cores needs to be developed. The aim is to generate designs that are optimised in terms of minimised power consumption, while providing acceptable application-level correctness and real-time performance. To this end, first the power minimisation problem is formulated and then a novel power minimisation technique through voltage scaling is developed.

5.4.1 Problem Formulation

Let P be the power of the multiprocessor system-on-chip (MPSoC) with C processing cores, Ω be the application-level correctness for the MPSoC application and D_r be the real-time performance (as decoding rate) of the application. The requirement is to reduce P through voltage scaling on processing cores of the MPSoC, such that application-level correctness, Ω, and real-time decoding rate, D_r, are maintained at acceptable levels (i.e.

$\Omega \geq \Omega_{ref}$ and $D_r \geq D_{rated}$). Since P is a function of power of processing cores due to voltage scaling, $P_i\,(V_{dd}, f)$, D_r is a function of operating frequency, f, and Ω is a function of increased soft error rate due to scaling of operating voltage, V_{dd}, the problem can be expressed as the following cost function minimisation:

$$\min : P = \sum_{i}^{C} P_i\,(V_{dd}, f) \quad , \tag{5.5}$$

subject to:

$$D_r\,(f) \geq D_{rated} \quad , \tag{5.6}$$

$$\Omega\,(V_{dd}) \geq \Omega_{ref} \quad . \tag{5.7}$$

Power minimisation through (5.5) with constraints (5.6) and (5.7) can be achieved using simulations, which requires trying every possible scaling option on each processing core in the worst case. For example, the MPEG-2 decoder (Figure 5.1(a)) with three different scaling options (Table 5.2) would require a total of $3^4 = 81$ simulations. This can be expensive with increasing number of processing cores and wider scaling options. To simplify the solution, a linear programming-based power minimisation through voltage scaling is proposed.

5.4.2 Power Minimisation

The dynamic power consumption, P, of an MPSoC with C cores and S scaling options on each core can be expressed as a function of binary scaling coefficient on each core, s_{ij} (s_{ij}=1 when j-th voltage scaling on the i-th processing core is selected and s_{ij}=0 when j-th voltage scaling on the i-th processing core is not selected), as

$$P = \sum_{i=1}^{C} \alpha_i \sum_{j=1}^{S} s_{ij} P_{ij} \quad , \tag{5.8}$$

where P_{ij} is the maximum power (P_{max}) of the i-th processing core for j-th scaling option (defined in (5.1)). Equation (5.8) can be used to define the required cost function (5.5) in terms of voltage scaling coefficient on each processing core as

$$\min : \; P = \sum_{i=1}^{C} \alpha_i \sum_{j=1}^{S} s_{ij} P_{ij} \quad . \tag{5.9}$$

To allow only one scaling option for a core from S scaling options in the cost function of (5.9), the necessary condition with s_{ij} is that the sum of all binary scaling coefficients

of a processing core should be equal to 1, i.e.

$$\forall_i; \sum_{j=1}^{S} s_{ij} = 1 \quad . \tag{5.10}$$

Equation (5.10) sets up the constraint related to choice of scaling options on each core within an MPSoC. The activity factor of the i-th processing core, α_i in (5.9), is given by ratio of processor execution time, T_i, to the multiprocessor execution time, T_M, i.e.

$$\alpha_i = \frac{T_i}{T_M} \quad . \tag{5.11}$$

The α_i, given by (5.11), is generally constant for a given application task mapping of an application with minor variation due to different operating conditions, including voltage and frequency scaling [165]. For power minimisation using the cost function definition in (5.9), the constraint according to (5.6) is the real-time performance, which needs to be greater than or equal to the MPEG decoding rate (in frames/s) in this case. For a video with total M frames and the total application time of T_A clock cycles (where $T_A = \sum_i T_i$), M/T_A frames are processed in each clock cycle and hence the decoding rate can be expressed as

$$D_r = \frac{M f_{eff}}{T_A} \quad , \tag{5.12}$$

where f_{eff} is the effective frequency experienced by multiprocessor application and is expressed as the sum of the chosen operating frequencies of the processing cores due to scaling, s_{ij}, weighted by their corresponding activity factors α_i, i.e.

$$f_{eff} = \sum_{i=1}^{C} \alpha_i \sum_{j=1}^{S} s_{ij} f_{ij} \quad , \tag{5.13}$$

where f_{ij} is the operating frequency of the i-th processing core due to j-th voltage scaling. Using the f_{eff} definition in (5.13), the decoding rate is given in terms of operating frequencies, f_{ij}, and scaling coefficient on the processing cores, s_{ij}, as

$$D_r = \frac{M}{T_A} \sum_{i=1}^{C} \alpha_i \sum_{j=1}^{S} s_{ij} f_{ij} \geq D_{rated} \quad . \tag{5.14}$$

The decoding rate expression in (5.14) sets up the constraint related to real-time performance as the target is to achieve a decoding rate greater than or equal to the real-time decoding rate, D_{rated}.

Next constraint described by (5.7) is related to the application-level correctness, which enables soft error-aware power minimisation. To develop such constraint, the requirement is that the overall SER of the multiprocessor, λ, should be limited such that acceptable application-level correctness can be achieved. Voltage scaling is an effective

technique to limit λ by controlling the SER of the processing cores. Let λ_i be the SER of the i-th processing core, R_i be the average register usage in bits per clock cycle and T_i be the execution time of the i-th processing core. The total number of SEUs experienced by i-th processing core is given by the product of the execution time (T_i), register usage (R_i) and the SER (in SEUs per bit per clock cycle) of the processing core (λ_i), i.e.

$$\Gamma_i = T_i R_i \lambda_i \quad . \tag{5.15}$$

Note that the λ_i in (5.15) is a function of the chosen voltage scaling coefficient on the processing core, i.e. $\lambda_i = s_{ij}\lambda_{ij}$, where λ_{ij} is the SER of i-th processing core with j-th scaling option. Hence, using (5.15) the total number of SEUs experienced by a decoder with C processing cores, Γ, is given as

$$\Gamma = \sum_{i=1}^{C} T_i R_i \left(\sum_{j=1}^{S} s_{ij}\lambda_{ij} \right) \quad . \tag{5.16}$$

Equation 5.16 gives the required expression for the total number of SEUs experienced in an MPSoC application with C processing cores. As λ_{ij} is the rate of SEUs per bit per clock cycle, the average number of SEUs per clock cycle that occur in the i-th processor is $R_i\lambda_{ij}$. Therefore, the average number of SEUs per clock cycle for the MPSoC is $\sum R_i\lambda_{ij}$. With overall register usage of $R_A = \sum R_i$, the SER per cycle for the MPSoC can be expressed as

$$\lambda = \sum_{i=1}^{C} \frac{R_i}{R_A} \sum_{j=1}^{S} s_{ij}\lambda_{ij} = \sum_{i=1}^{C} \rho_i \sum_{j=1}^{S} s_{ij}\lambda_{ij} \quad , \tag{5.17}$$

where $\rho_i = R_i/R_A$, which is generally constant for a given application and architecture [172]. Equation (5.17) gives the relationship between SER, λ_{ij}, and voltage scaling of processing cores within an MPSoC. To facilitate power minimisation through voltage scaling considering the impact on the application-level correctness (Ω), (5.17) needs to be extended to incorporate the relationship between voltage scaling and Ω. In the following section, such relationship is found through statistical sampling and curve-fitting technique to introduce application-level correctness (Ω) constraint in the proposed linear programming-based power minimisation technique.

5.4.3 Application-level Correctness and Voltage Scaling Relationship

This section introduces the first investigation into the relationship between application-level correctness and power minimisation through voltage scaling. The objective is to enable the proposed voltage scaling technique to generate designs with minimised power consumption, whilst meeting an acceptable application-level correctness and specified

real-time performance using this relationship. To establish such relationship, the effect of voltage scaling on each processing core on the application-level correctness is first investigated, which is referred to as the application sensitivity of each processing core within the MPSoC. According to [173], such sensitivity (in terms of reliability) of a multiprocessor system consisting of a number of processing elements is related to the number of soft errors experienced over the application time and the nature of computation being carried out. The total number of SEUs experienced by the i-th processing core, Γ_i, over the application time for a given SER, λ_i, is given by (5.15). Since λ_i depends on the voltage scaling being used due to (5.4), from (5.15) it is evident that the sensitivity of the i-th processing core in terms of the number of SEUs experienced (Γ_i) depends on voltage scaling, execution time (T_i), and average register usage (R_i) of the processing core. To demonstrate the sensitivity of the processing cores within the MPSoC decoder (Figure 5.1(a)) for a given SER, Table 5.4 shows the execution time, T_i (in cycles) and the register usage, R_i (in kbits/cycle) of each processing core obtained through simulations. The multiprocessor execution time, T_M (in cycles), of the decoder is shown in row

Core	**Execution Time (T_i)**, *cycles*	**Register Usage (R_i)**, *kbits/cycle*
VLD	8.04×10^8	23.1
ISQ	6.49×10^8	19.2
IDCT	1.17×10^9	19.3
MC	8.62×10^8	25.3
Multiprocessor	T_M=(1.407×10^9)	–
Total	T_A=(3.48×10^9)	R_A=87

TABLE 5.4: Execution time and register usages of the processing cores of the MPEG-2 video decoder with four processing cores (Figure 5.1(a))

6, while the application time, T_A (the sum of execution times of all processing cores, i.e. $T_A = \sum_i T_i$, in cycles) and total register usage, R_A (in kbits/cycle) are shown in row 7 (Table 5.4). Using the T_i and R_i values from Table 5.4 in (5.15), and assuming SER of 3.98×10^{-8} (SEUs per bit per clock cycle) for no voltage scaling, the processing cores VLD, ISQ, IDCT and MC of the decoder (Figure 5.1(a)) experience 7.4×10^5, 4.8×10^5, 8.7×10^6 and 8.3×10^5 SEUs, respectively. Hence, the most sensitive task of the decoder in terms of largest number of SEUs experienced (Γ_i) is the IDCT, followed by the tasks MC, VLD and ISQ. To evaluate the corresponding sensitivity at the application-level, firstly the decoder PSNR performance (Ω) over varying λ_0 is observed when the supply voltages of all processing cores are scaled by the same amount. Also, to observe the application-level sensitivity of each core, the PSNR performance (Ω) over varying λ_0 is obtained, when SEU injection is carried out one core at a time. Figure 5.5 shows the corresponding PSNR values (Ω) for different scaling options. The Ω values are obtained from average of 100 iterations of decoding the *tennis* video sequence (Table 5.1) for a

specified λ_0 and varying λ_0 over the dynamic range from 10^{-10} to 10^{-5}. Each fault injection experiment took on average 112 seconds for each iteration on Intel(R) Core(TM)2 2GHz CPU running SystemC in RHEL4.

From Figure 5.5(a)-(c), three key observations are made as follows:

Observation 1 : Comparing the PSNR values (Ω, in dB) from Figure 5.5(a)-(c), it is evident that the MPEG-2 decoder has the worst application-level correctness (i.e. the lowest PSNR) when the voltage of all the four processing cores are scaled by s=3. This is due to the increase in soft error rate with such aggressive voltage scaling to reduce power as described in Section 5.3.

Observation 2 : The second observation is related to the sensitivity of each task of the decoder. From Figure 5.5(a), it can be seen that the task IDCT is the most sensitive in terms of application-level correctness, followed by the tasks MC, VLD and ISQ. The highest sensitivity of the task IDCT is because it experiences the highest number of SEUs (given by (5.15) and Table 5.4) and carries out processing that is more susceptible to SEUs, such as video block-level multiplication and transpose. The other tasks experience less SEUs and carry out video processing that are less susceptible to SEUs, such as header decoding in the task VLD and video block-level substitution and fixed point multiplications in the tasks MC and ISQ. It can be seen from Figure 5.5(b) and 5.5(c) that as the supply voltage is scaled by a factor of 2 or 3, the PSNR (Ω) decreases significantly due to the increase in SER. For example, at $\lambda_0 = 3.98 \times 10^{-8}$, when the voltage scaling is reduced from 1V to 0.44V (s=3), the number of SEUs experienced by the IDCT core increases from 8.7×10^5 to 27×10^5, causing the degradation of the overall PSNR (Ω) from $75dB$ (no scaling) to $42dB$ (scaling by 3).

Observation 3 : Referring to Figure 5.5(a), it can be seen that the PSNR values (Ω, given by (5.3)) decrease with increasing SERs. The trend of reduction of the PSNR values exhibit approximate exponential degradation with increasing SERs until about $20dB$. Beyond this point, the PSNR degradation is approximately linear in the dB scale. This phenomenon can be explained as follows. With higher SER, further PSNR degradation normally takes place with the predicted video frames within a group of pictures of a video sequence. However, the intra-coded or unpredicted video frames are decoded with rather higher PSNR (Ω) compared to predicted video frames. This is because the decoding of unpredicted frames are much simpler due to no or less compression and motion compensation, while the predicted frames require complex decoding process involving decompression and motion compensation. The higher complexity in processing makes predicted frames highly sensitive in the presence of SEUs [141]. The cumulative effect is seen as an approximate linear degradation below $25dB$ PSNR with higher SERs.

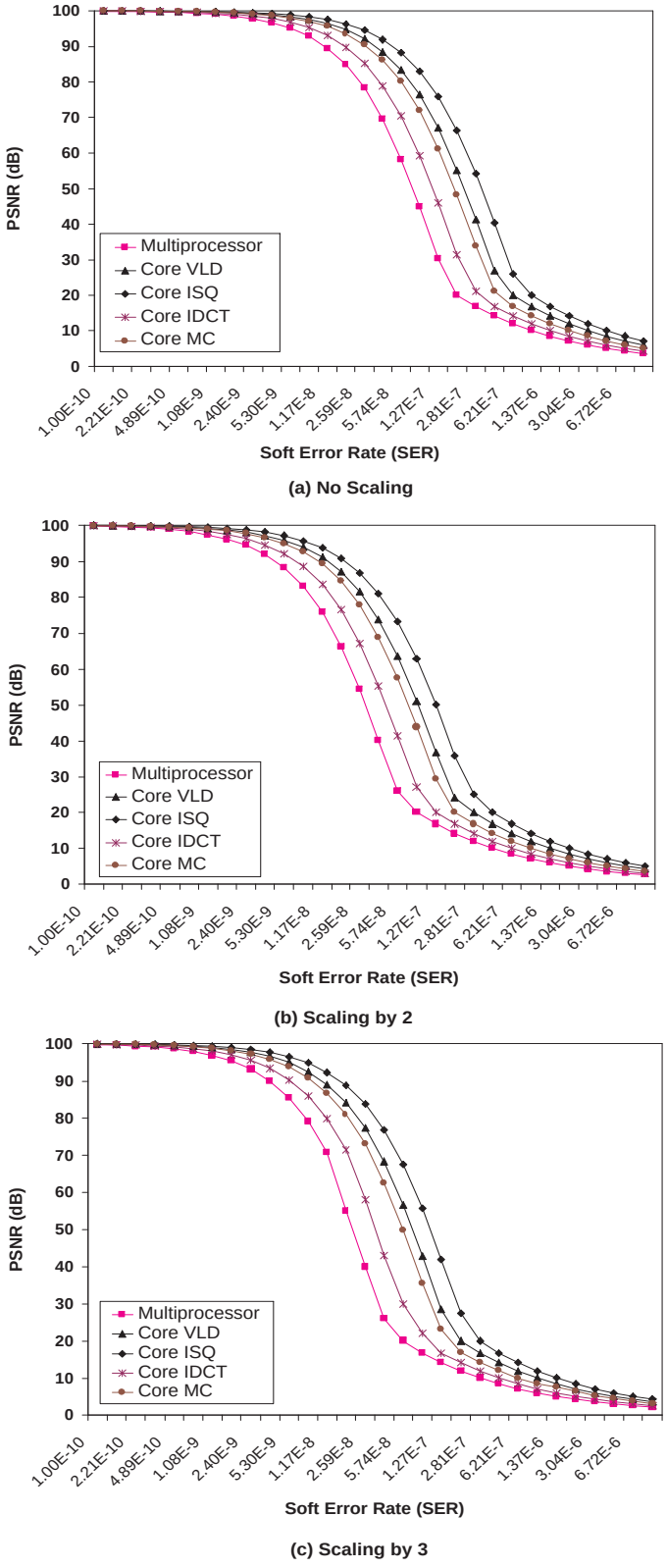


FIGURE 5.5: Application sensitivity (in terms of PSNR, in dB) of processing cores and multiprocessor MPEG-2 video decoder with varying base soft error rate, λ_0, with (a) no voltage scaling, (b) voltage scaling by 2, and (c) voltage scaling by 3

Following the observation relating to application sensitivity and PSNR variation (Figure 5.5), an approximate relationship between application-level correctness in terms of PSNR (Ω) and SER (λ) is found next. From Figure 5.5(a)-(c), using curve-fitting technique, an expression for PSNR (from $100dB$ to $20dB$, which would suffice for the $30dB$ acceptable lower limit), Ω, in terms of SER, λ, is found as

$$\Omega = 100 \exp\left(-a\lambda\right), \tag{5.18}$$

where a is the exponential decay constant. Empirically, the a values were found with regression coefficient of 95% or more closeness of approximation. For example, the curve-fitting of the decoder PSNR shown in Figure 5.5(a) gives a value of 11.96×10^6, resulting in a regression coefficient of 98.2% closeness of approximation. Now, from (5.18), SER can be given as, $\lambda = \left[\frac{1}{a} \ln\left(\frac{100}{\Omega}\right)\right]$. Substituting this λ expression in (5.17) yields:

$$\sum_{i=1}^{C} \rho_i \sum_{j=1}^{S} s_{ij} \left(\frac{1}{a_{ij}}\right) \ln\left(\frac{100}{\Omega_{ij}}\right) = \frac{1}{a} \ln\left(\frac{100}{\Omega}\right) \quad , \tag{5.19}$$

where a_{ij} is the exponential decay constant for i-th decoder core with j-th scaling option, Ω is the PSNR of multiprocessor decoder and Ω_{ij} is the PSNR of the i-th decoder core with j-th scaling option. Equation (5.19) gives the required relationship between application-level correctness and voltage scaling on the processing cores within an MPSoC. To provide a better insight into the impact of V_{dd} scaling, power consumption and PSNR values for different voltage scalings are shown in Figure 5.6(a) and (b) employing (5.8), (5.14) and (5.19) for the *tennis* video sequence at SER of 3.98×10^{-8}. The voltage scaling coefficients are shown in horizontal axes, where the four digits represent voltage scaling coefficients of processing cores IDCT, MC, VLD and ISQ, respectively. As can be seen, voltage scaling with higher coefficients achieve higher power reduction (Figure 5.6(a)). However, this also affects the application-level correctness of the MPEG-2 decoder in terms of PSNR, giving unacceptable application-level correctness (Figure 5.6(b)).

To set the application-level correctness constraint through (5.19), the necessary condition is to limit the overall SER such that it gives the application-level correctness at or above the acceptable level, i.e. $\Omega_{ref} \geq 30dB$ (Section 5.2.4). Considering the overall SER produced as a result of the chosen voltage scaling options on the processing cores (given by (5.17)) is applied to multiprocessor decoder cores with no scaling, (5.19) can be expressed in terms of Ω_{ref} to give the following inequality expression:

$$\sum_{i=1}^{C} \rho_i \sum_{j=1}^{S} s_{ij} \left(\frac{1}{a_{ij}}\right) \ln\left(\frac{100}{\Omega_{ij}}\right) \leq \frac{1}{a_{j=1}} \ln\left(\frac{100}{\Omega_{ref}}\right) \quad . \tag{5.20}$$

Equation (5.20) defines the final constraint based on application-level correctness. The

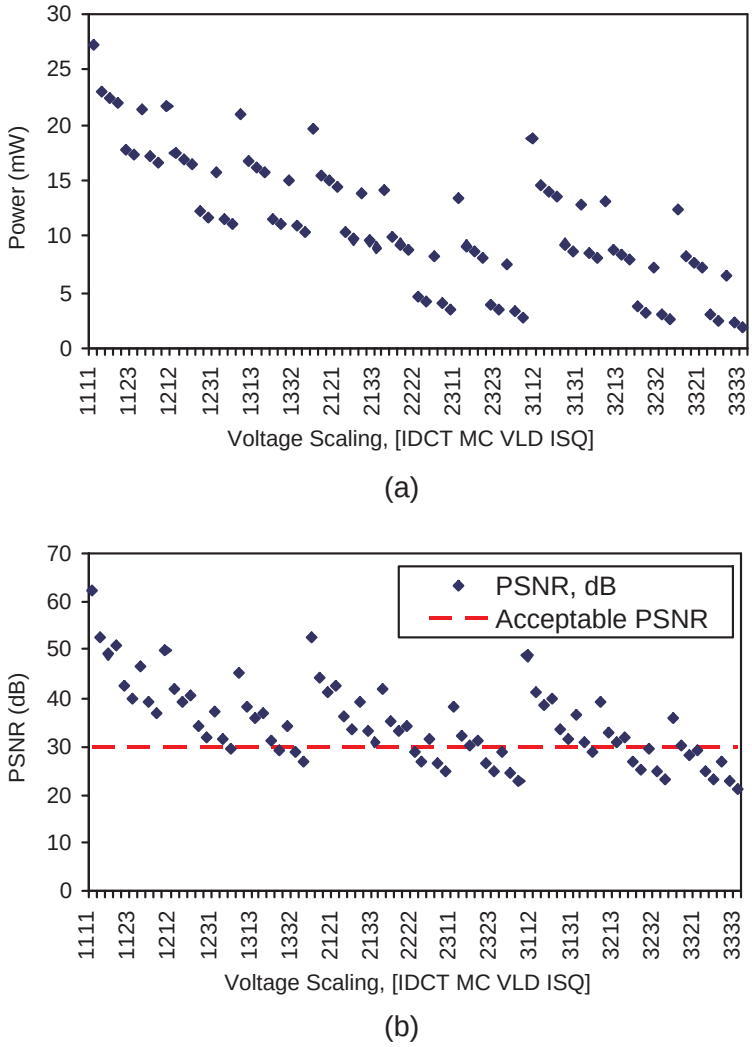


FIGURE 5.6: (a) Power consumption (P, in mW) for different voltage scaling options using *tennis* video sequence at soft error rate (SER) of 3.98×10^{-8}, and (b) PSNR values (Ω, in dB) for different voltage scaling options using *tennis* video sequence at soft error rate (SER) of 3.98×10^{-8}

proposed voltage scaling technique employs linear programming to solve for optimised voltage scaling of each core within an MPSoC architecture and gives minimised power using cost function in (5.9) and constraint functions in (5.10), (5.14) and (5.20).

5.5 Experimental Results

In this section, the effectiveness of the proposed voltage scaling technique (Section 5.4) for power minimisation is evaluated using the MPEG-2 video decoder (Figure 5.1(a)) as a case study, whilst providing acceptable application-level correctness ($30dB$ PSNR as also used by [168, 169]) and real-time performance (Table 5.1). In this section, the proposed soft error-aware voltage scaling technique is illustrated and compared with the

soft error-unaware techniques. Also, the impact of choice of scaling levels and acceptable level of application-level correctness on power minimisation using the proposed voltage scaling technique is presented. Table 5.5 shows three experiments carried out using the *tennis* video sequence, denoted as Exp:1, Exp:2 and Exp:3. Cycle-accurate simulator MPARM [21] is used to obtain the reported power consumption values (further details can be found in Section B.2, Appendix B), while fault injection is carried out using the fault injection simulator (Section 5.2.3). The three experiments are explained in the following:

Exp:1 shows three aggressive voltage scaling options for the decoder cores: with no scaling (i.e. $s = 1$) or with the same voltage scaling coefficients (i.e. $s = 2$ and $s = 3$) at SER of 3.98×10^{-8} (rows 2, 3 and 4, Table 5.5). As expected, power consumption is reduced when lower voltage scaling is applied compared to no voltage scaling. This reduction in power consumption is achieved in some cases with unacceptable PSNR and decoding rate. For example, when $s = 3$, power consumption is lowest but PSNR is $23dB$ (less than $30dB$ acceptable PSNR) and decoding rate 19 frames/s (less than 29 frames/s, Table 5.1).

Exp.	**SER, λ** ($\times 10^{-8}$)	*Scaling Coefficients*				P, mW	Ω, dB	D_r, *frames/s*
		Core VLD	**Core ISQ**	**Core IDCT**	**Core MC**			
Exp:1	3.98	1	1	1	1	27.2	65	65
	3.98	2	2	2	2	4.9	32	32
	3.98	3	3	3	3	2.5	23	19
Exp:2	3.98	2	3	2	2	4.2	31	29
	5.75	2	1	2	2	8.9	33	31
	8.53	1	1	2	1	19.7	32	51
Exp:3	3.98	2	3	2	2	4.2	31	29
	5.75	2	3	2	2	4.2	23	29
	8.53	2	3	2	2	4.2	18	29

TABLE 5.5: Application-level correctness and power minimisation trade-off using *tennis* video sequence

Exp:2 demonstrates the effectiveness of the proposed voltage scaling technique to generate designs with minimised power consumption that maintains acceptable application-level correctness and meets the real-time performance constraint for three given SERs (rows 5, 6 and 7, Table 5.5). As can be seen, at SER of 3.98×10^{-8}, three out of four decoder cores have been scaled by 2 (f=100MHz and V_{dd}=0.58V), while the other core (ISQ) has been scaled by 3 (f=66.67MHz and V_{dd}=0.44V). As a result of such voltage scaling power is reduced to 4.2mW, while maintaining PSNR of $31dB$ ($>30dB$ PSNR) and decoding rate of 29 frames/s. As SER is increased to 5.75×10^{-8}, the core ISQ

could not scaled, while the other cores were scaled by 2 to produce design with acceptable PSNR. Due to increased voltage scaling, power consumption increases to 8.9mW to maintain the specified application-level correctness (>30dB PSNR) and decoding rate (29 frames/s) compared to 4.2mW for the case when SER is 3.98×10^{-8}. The reason for core ISQ of the decoder not being scaled is that the core has low power contribution due to low activity factor (Section 5.4.3) and also less sensitivity to application-level correctness (Figure 5.5). The low sensitivity of core ISQ in terms of application-level correctness arises from video block-level substitution and quantisation, which are also less affected by the lower number of SEUs injected (Section 5.4.3). With further increase in SER to 8.53×10^{-8}, only one of the four processing cores (core IDCT) have been scaled by 2 to meet acceptable PSNR, leading to even higher power consumption of 19.7mW. Comparing between Exp:2 and Exp:1 at SER of 3.98×10^{-8} (Table 5.5), the proposed voltage technique reduces the power consumption by 85%, while ensuring the application-level correctness (Ω) and real-time decoding performance (D_r) are at acceptable levels. These two experiments demonstrate the trade-off between application-level correctness in terms of PSNR and power consumption that exists when generating designs with reduced power consumption and acceptable levels of PSNR and real-time performance.

Exp:3 highlights the significance of considering application-level correctness constraint for power minimisation using a soft error-unaware voltage scaling technique. The aim is to show that using such voltage scaling technique it is possible to produce designs with low power consumption that violate application-level correctness requirement. Numerous soft error-unaware voltage scaling for power minimisation has been proposed to date, most of which employ the principles of workload balancing and allocation, such as [52, 174]. Based on these principles, Exp:3 shows a power minimisation technique such that processing cores with high activity factors (i.e. higher workload) can operate with no or higher voltage scaling and processing cores with low activity factors (i.e. lower workload) can operate with lower voltage scaling. As can be seen from Exp:3 (Table 5.5), using soft error-unaware technique with the SERs of 5.75×10^{-8} and 3.98×10^{-8} give similar voltage scaling but results in an unacceptable application-level correctness of 23dB PSNR for higher SER and even lower PSNR with increasing SERs. The above three experiments were carried out using GNU Linear Programming Kit (GLPK) to solve for the linear programming-based power minimisation. The average solution time for a set of voltage scaling coefficients using GLPK is between 2-10 seconds on an Intel(R) Core(TM)2 2GHz CPU running RHEL4.

In Table 5.5, the experimental results have been obtained using the *tennis* video sequence. Other video sequences given in Table 5.1 have also been experimented using Exp:2. Similar trade-off between application-level correctness and power consumption were observed as shown in Table 5.6. As can be seen, the decoder (Figure 5.1(a)) processing *house* video sequence with smaller resolution of frames (i.e. QCIF resolu-

Video Seq.	**SER** ($\times 10^{-8}$)	*Scaling Coefficients*				**P**, mW	Ω, dB	D_r, *frames/s*
		Core VLD	**Core ISQ**	**Core IDCT**	**Core MC**			
house	3.98	2	2	3	3	3	30	98
	5.75	2	1	2	1	8.9	30	175
	8.53	1	1	2	1	19.7	31	204
flower	3.98	1	1	2	1	19.7	47	26
	5.75	1	1	1	2	21.6	32	28
	8.53	1	1	1	1	27.2	31	32

TABLE 5.6: Application-level correctness and power minimisation trade-off using other video sequences

tion, Table 5.1) gives lower power consumption of 3mW compared to 4.2mW for *tennis* video sequence, while maintaining acceptable level of application-level correctness (30dB PSNR) and real-time decoding rate (98 frames/s). This is because at lower resolutions, the decoder requires less number of clock cycles per frame (for example, 2×10^6 clock cycles per frame for *house* video sequence, compared to 7.95×10^6 clock cycles per frame for *tennis* video sequence), making it easier to achieve decoding rate constraint given by (5.14). As the frame resolution increases, the decoder requires more processing cycles per frame (for example, 15.5×10^6 clock cycles per frame for *flower* video sequence) and gives less opportunities for voltage scaling. This gives higher power consumption of 19.7mW for *flower* video sequence compared to 4.2mW for *tennis* video sequence. With increased SER, the proposed voltage scaling technique limits the voltage scaling options to maintain acceptable level of application-level correctness, resulting in higher power consumption (Table 5.6).

The choice of number of available scaling levels has a crucial role in power minimisation using the proposed voltage scaling technique. In Exp:2 (Table 5.5), only three levels of scaling (Table 5.2) was used. To demonstrate the impact of choice of scaling option using proposed voltage scaling, Table 5.7 shows the power consumption (in mW), PSNRs (in dB) and decoding rates (in frames/s) for S=2 (with s=1 and s=2 only in Table 5.2) at different SERs. The results in Table 5.7 are generated using *tennis* video sequence (Table 5.1) with different SERs. As can be seen from Table 5.7, with scaling option of S=2, the solution space for the cost function of the linear programming-based proposed voltage scaling technique shrinks for a given number of processing cores (according to (5.9)), giving it less flexibility of scaling voltages of each core and minimizing power further. For example, at SER of 3.98×10^{-8}, the proposed voltage technique gives power consumption of 4.9mW for S=2 compared to 4.2mW for S=3 (Exp:2, Table 5.5). As expected, the power consumption increases further with increased SER as voltage scaling options become limited for the given application-level correctness ($\Omega \geq 30dB$) and

SER ($\times 10^{-8}$)	*Scaling Coefficients*				**P**, mW	Ω, dB	D_r, $frames/s$
	Core VLD	**Core ISQ**	**Core IDCT**	**Core MC**			
3.98	2	2	2	2	4.9	32	32
5.75	2	1	2	2	8.9	33	31
8.53	1	1	2	1	19.7	32	51

TABLE 5.7: Application-level correctness and power minimisation trade-off using number of voltage scaling levels of S=2

decoding rate constraints ($D_r \geq 29 fps$).

In Exp:2 (Table 5.5), $30dB$ has been used as the minimum acceptable application-level correctness, which is also used by [169]. To investigate the impact of choice of acceptable application-level correctness in terms of PSNR, $20dB$ (as suggested by [175]) and $40dB$ (as suggested by [176]) PSNRs are also used as constraints in (5.7). Table 5.8 shows the power consumption (in mW), PSNRs (in dB), real-time decoding rates (in frames/s) for these constraints. The results in Table 5.8 are generated using *tennis* video sequence of Table 5.1 and using the SER of 8.53×10^{-8}. As can be seen in Table 5.8, due to

Acceptable PSNR, dB	*Scaling Coefficients*				**P**, mW	Ω, dB	D_r, $frames/s$
	Core VLD	**Core ISQ**	**Core IDCT**	**Core MC**			
20	2	2	2	2	8.9	20	32
40	1	2	1	1	24.5	42	62

TABLE 5.8: Power consumption (in mW), PSNRs and decoding rates of the MPEG-2 video decoder (Figure 5.1(a)) using the proposed voltage scaling with different levels of acceptable application-level correctness (in terms of PSNR, dB)

lowered requirement of acceptable application-level correctness, such as $20dB$ PSNR (row 2), the proposed voltage scaling technique gives minimised power consumption of 8.9mW compared to 19.7mW for acceptable PSNR of $30dB$ (row 4, Table 5.5) with the same SER. This is because with such low requirement of acceptable PSNR, achieving the application-level correctness constraint in (5.20) becomes easier with higher scaling options. On the other hand, when acceptable PSNR is set to a higher level, achieving power minimisation with higher scaling becomes harder and the proposed voltage scaling technique gives higher power consumption. For example, with acceptable PSNR of $40dB$, the proposed voltage scaling gives no scaling on all cores except for the core ISQ. This results in a higher power consumption of 24.5mW (row 3, Table 5.8), compared to 19.7mW for acceptable PSNR of $30dB$ (row 4, Table 5.5).

The experimental results in this section have been obtained using MPSoC decoder architecture with four processing cores (Figure 5.1(a)), while decoding different video

sequences (Table 5.1). A number of trade-offs between application-level correctness and power consumption using voltage scaling have been observed. In the following section, the effect of application task mapping (distribution of tasks among cores of the MPSoC architecture) and architecture allocation (choice of the number of cores needed in MPSoC architecture) on power minimisation using the proposed voltage scaling technique is investigated, while maintaining acceptable application-level correctness and real-time performance.

5.6 Application Task Mapping and Architecture Allocation

The influence of application task mapping and architecture allocation on system performance in the context of HW/SW co-design has been investigated extensively [48, 147, 177]. In this section, the impact of application task mapping and architecture allocation on the trade-off between application-level correctness and power minimisation is investigated.

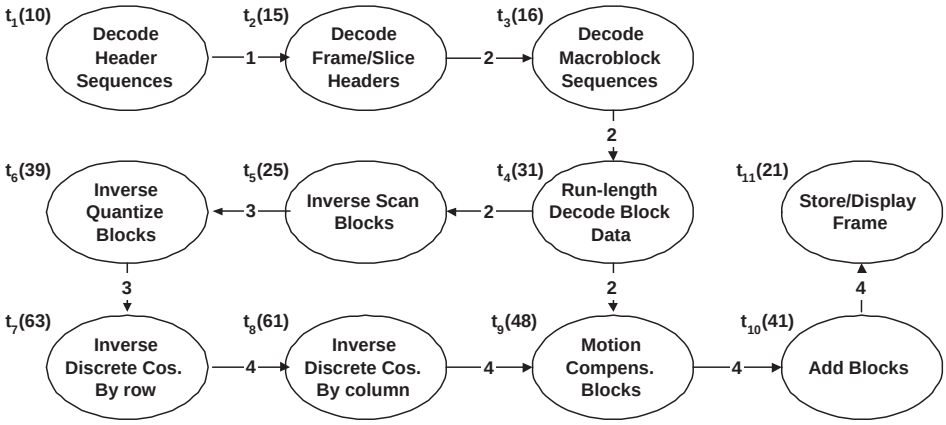


FIGURE 5.7: Task graph of MPEG-2 video decoder (Figure 5.1(a)) with eleven tasks

5.6.1 Application Task Mapping

Application task mapping is a crucial step in the design of the MPSoC applications, which involves distribution of the computation and communication tasks among the processing cores and interconnects of an MPSoC architecture. Figure 5.7 shows the task graph of the MPEG-2 video decoder (Figure 5.1(a)) with eleven tasks. Each node represents a computational task weighted by number in parenthesis, indicating the cost in terms of execution time. The edge between nodes represents the communication task shown with cost that describes the time required to transfer data between the tasks with shown directions. All costs are multiples of 5.5×10^6 clock cycles and are

obtained through SystemC cycle-accurate simulations assuming 32-bit transfer width. The computational tasks are modelled as separate task processes, while the communication between tasks is modelled as message passing queues. The communication time between tasks is found by dividing the size of inter-task queue by the bandwidth of the channel (in bits per cycle). The effect of mapping the tasks on processing cores on communication cost is not modelled explicitly rather an worst-case approximation is assumed. Similar assumptions have also been used in [74, 172].

Mapping	**Core**	**Mapped Tasks**	R_i, *kbits/cyc.*	T_i, cyc. $_{(\times 10^9)}$
M1 (Figure 5.1(a))	Core 1	t_1, t_2, t_3, t_4	23.1	0.804
	Core 2	t_5, t_6	19.3	0.649
	Core 3	t_7, t_8	19.4	1.165
	Core 4	t_9, t_{10}, t_{11}	25.4	0.862
	Multiprocessor		87	1.407
M2 (optimised for reduced register usage)	Core 1	t_1, t_2, t_3, t_4, t_5, t_6	25.6	1.254
	Core 2	t_7, t_8	19.4	1.2098
	Core 3	t_9, t_{10}	21.2	0.8317
	Core 4	t_{11}	14.2	0.1845
	Multiprocessor		80	1.489
M3 (optimised for parallelism)	Core 1	t_1, t_2, t_3, t_4, t_5	23.1	0.9077
	Core 2	t_6, t_7	23.1	0.9835
	Core 3	t_8	19.4	0.6052
	Core 4	t_9, t_{10}, t_{11}	25.4	0.9835
	Multiprocessor		91	1.258
M4 (optimised for reduced register usage & parallelism)	Core 1	t_1, t_2, t_3, t_4, t_5	23.2	0.9078
	Core 2	t_6, t_7	23.2	0.9835
	Core 3	t_8, t_9	20.1	1.031
	Core 4	t_{10}, t_{11}	23.9	0.5577
	Multiprocessor		88	1.261

TABLE 5.9: Different task mappings, register usages and execution times for MPSoC decoder using four cores

Numerous mapping combinations are possible for decoder design using the MPEG-2 video decoder task graph (Figure 5.7) on the MPSoC architecture (Figure 5.1). Table 5.9 shows four different task mappings carried out based on the decoder task graph (Figure 5.7). The mapping M1 is the mapping employed in Figure 5.1(a), M2 is mapping optimised for reduced register usage, M3 is mapping optimised for high parallelism and finally, M4 is mapping jointly optimised for reduced register usage and high par-

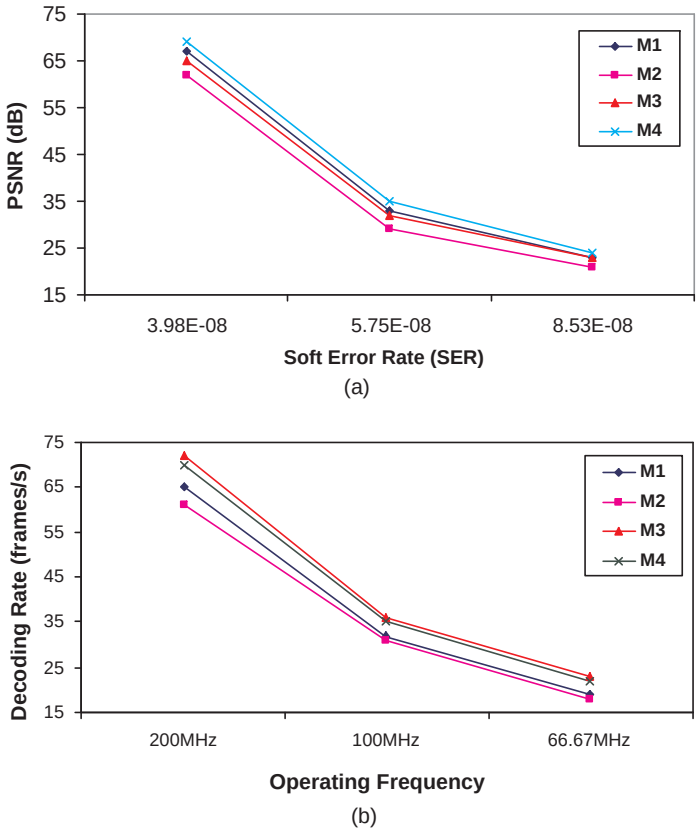


FIGURE 5.8: Impact of different application task mappings on (a) application-level correctness and (b) decoding rates of the MPEG-2 video decoder

allelism. The task mappings M2, M3 and M4 in Table 5.9 are found by simulated annealing technique using group migration-based task movement proposed in [172]. The simulated annealing is carried out with adaptive simulated annealing tool [178] (refer to Appendix B for more details) using the parameters reported in [172]. As expected, mapping M2 has the lowest register usage (80 kbits/cycle) due to localisation of tasks (more tasks mapped in a processing core) and mapping M3 has the lowest multiprocessor execution time due to high parallelism among processing cores (1.258×10^9 clock cycles). Mapping M4 offers a good trade-off between register usage and multiprocessor execution time by carefully distributing the tasks among processing cores. Figure 5.8(a) shows the impact of application task mapping on application-level correctness in terms of PSNR for different SERs. As expected, the PSNR worsens as the SER increases for all mappings. Mapping M2 exhibits the worst PSNR ($62dB$) when compared with other three mappings (Figure 5.8(a)). This is because, reduced register usage in M2 is achieved through localization of tasks, which eventually increases the multiprocessor execution time and gives higher number of SEUs experienced, given by (5.16). Due to increased multiprocessor execution time, mapping M2 gives the lowest decoding rate (62 frames/s) for a given given operating frequency (Figure 5.8(b)). Mapping M3 gives the best decoding rate (73 frames/s) as tasks are mapped to give high parallelism. It can be

seen that mapping M4 gives the highest PSNR ($71dB$) among all task mappings. The highest PSNR in mapping M4 is due the lowest SEUs experienced, given by (5.16). To demonstrate the trade-off between power consumption and application-level correctness for different task mappings (M2, M3 and M4), Table 5.10 shows the voltage scaling of the decoder processing cores. As can be seen, mapping M4 has the lowest power con-

Mapping	**SER,** λ_0 ($\times 10^{-8}$)	*Scaling Coefficients*				P, mW	Ω, dB
		Core 1	**Core 2**	**Core 3**	**Core 4**		
M2	3.98	2	2	2	1	5.6	35
	5.75	2	2	1	1	10.7	32
	8.53	1	1	1	1	26	37
M3	3.98	2	3	2	2	4.8	30
	5.75	2	2	1	2	9.9	33
	8.53	1	1	1	2	20	32
M4	3.98	2	3	3	2	3.6	31
	5.75	2	2	2	2	5.5	30
	8.53	1	1	2	1	19.1	30

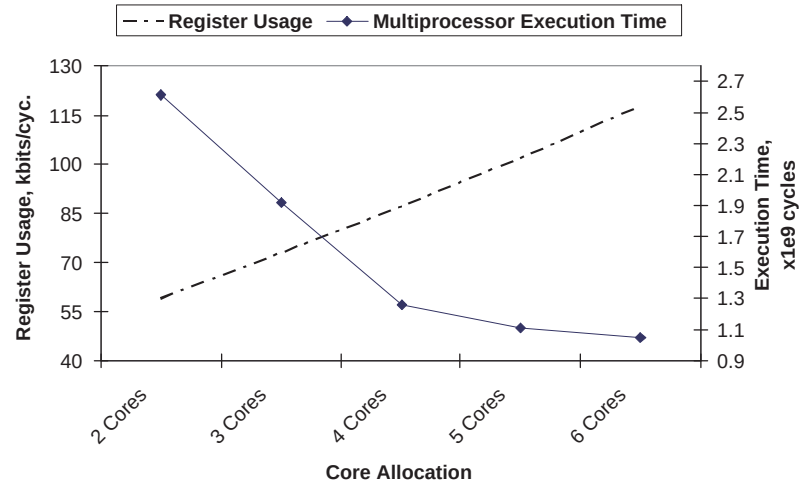
TABLE 5.10: Impact of decoder task mapping on the power minimisation using the proposed voltage scaling technique

sumption for different SERs, while it provides acceptable PSNR values ($\geq 30dB$ PSNR) when compared with the other three mappings (M1 mapping results are reported in Exp:2, Table 5.5). Due to joint optimisation with high multiprocessor parallelism and low register usage in mapping M4, it gives more opportunities for voltage scaling for power minimisation using the proposed technique and saves 16.7% power than mapping M1, 55.6% power than mapping M2 and 33.3% power than mapping M3 at SER of $\lambda_0 = 3.98 \times 10^{-8}$ (Table 5.10). Due to the lowest PSNR in mapping M2 (Figure 5.8), there is less opportunity for voltage scaling on the processing cores. Hence, mapping M2 gives the highest power consumption among all task mappings. As the SER increases, voltage scaling is limited but similar trade-off between power consumption and application-level correctness is observed. From Table 5.10, it is evident that the task mapping that is jointly optimised for reduced register usage and reduced multiprocessor execution time (mapping M4) offers the best choice among all mappings (Table 5.9) for power minimisation using the proposed technique, while meeting a real-time performance and maintaining an acceptable application-level correctness.

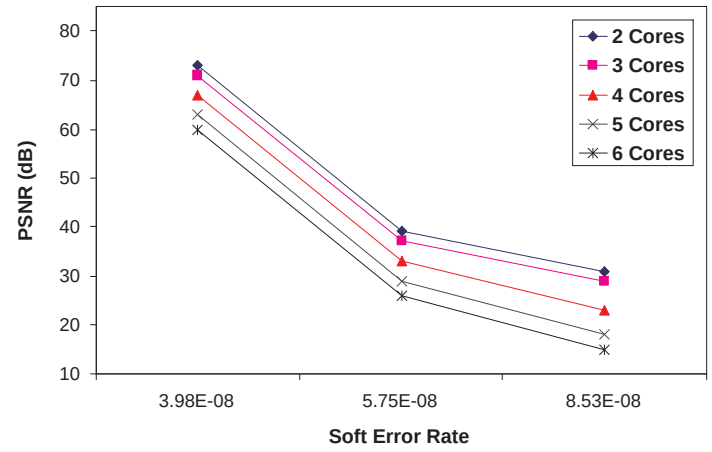
5.6.2 Architecture Allocation

Architecture allocation is a system-level design step for MPSoCs involving the choice of allocation of different components and their interconnections into the architecture (see Section 2.4.2.1, Chapter 2 for further details). In this work, architecture allocation is referred to as allocation of number of processing cores in the MPSoC architecture. The impact of architecture allocation on power minimisation and application-level correctness is investigated next.

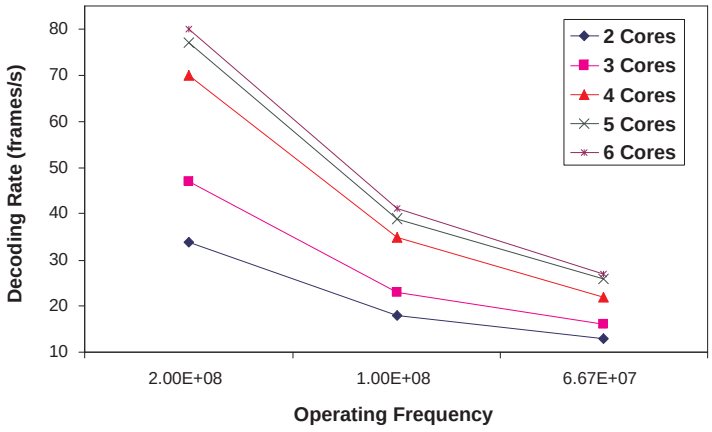
Architecture allocation with higher number of processing cores allows for more computational resources and higher parallelism. For the MPSoC decoder increasing the number of processing cores is expected to give lower multiprocessor execution time and better decoding rate. However, with higher number of processing cores tasks distribution causes shared memory resources to get duplicated further, increasing the register usage of the MPSoC application. The higher register usage, in turn, increases the total number of SEUs seen by the application (given by (5.15)), the application-level correctness becomes worse. Hence, an interesting problem is to choose the right number of allocated cores in an architecture for the multiprocessor application that gives minimum power consumption using the proposed voltage scaling technique, while meeting the real-time performance and maintains acceptable application-level correctness.



(a)



(b)



(c)

FIGURE 5.9: (a) Multiprocessor execution time and register usage for different architecture allocations, (b) PSNRs for different architecture allocations at different SERs, and (c) Decoding rates of different architecture allocations at different operational frequencies

Allocation	**Core**	**Mapped Tasks**
2 Cores	Core 1	t_1, t_2, t_3, t_4, t_5, t_6, t_7
	Core 2	t_8, t_9, t_{10}, t_{11}
3 Cores	Core 1	t_1, t_2, t_3, t_4, t_5, t_6
	Core 2	t_7, t_8
	Core 3	t_9, t_{10}, t_{11}
4 Cores	Core 1	t_1, t_2, t_3, t_4, t_5
	Core 2	t_6, t_7
	Core 3	t_8, t_9
	Core 4	t_{10}, t_{11}
5 Cores	Core 1	t_1, t_2, t_3, t_4
	Core 2	t_5, t_6
	Core 3	t_7
	Core 4	t_8, t_9
	Core 5	t_{10}, t_{11}
6 Cores	Core 1	t_1, t_2, t_3, t_4
	Core 2	t_5, t_6
	Core 3	t_7
	Core 4	t_8
	Core 5	t_9
	Core 6	t_{10}, t_{11}

TABLE 5.11: Task distribution among cores for different architecture allocations using MPEG-2 video decoder task graph (Figure 5.7)

To demonstrate the impact of architecture allocation on task mapping, Table 5.11 shows the mapped tasks for different allocations from 2 cores to 6 cores. The task mappings are carried out using MPEG-2 video decoder task graph (Figure 5.7) using joint optimisation with reduced register usage and high parallelism (mapping M4) as it has been shown to effectively reduce power consumption through the proposed voltage scaling technique (Section 5.6.1). The architecture allocation and per core mapped tasks are shown in columns 1-3 (Table 5.11). To demonstrate the impact of architecture allocation on application-level correctness, Figure 5.9(a) shows the register usage (in kbits per cycle) and multiprocessor execution times (in 10^9 cycles) for different decoder architectures (from 2 cores to 6 cores) with mapped tasks shown in Table 5.11. As expected, as the number of cores increases the register usage also increases due to duplication of task memory resources among cores. Also, the multiprocessor execution time decreases due to higher parallelism among processing cores. Figure 5.9(b) shows the resultant PSNRs (in dB) for different SERs using *tennis* video sequence and Figure 5.9(c) shows the decoding rates for different operating frequencies. As expected, architecture with 2 cores has the highest PSNR (Figure 5.9(b)) due to lowest register usage and subsequently the least number of SEUs injected (according to (5.15)). However, due to higher multiprocessor execution time, architecture with 2 cores has the the lowest decoding rate (Figure 5.9(c)) when compared with the other architectures. With increased number of allocated cores (i.e. from 2 cores to 6 cores), PSNR decreases with increased register usage (and hence increased number of SEUs experienced), while the decoding rate increases due to decreased multiprocessor execution time. For example, architecture with 6 cores give the lowest PSNR (Figure 5.9(a)) and highest decoding rates (Figure 5.9(b)) compared to other architectures.

To demonstrate the trade-off between power consumption and application-level correctness for different architecture allocations, Table 5.12 shows the power consumption (in mW), PSNR (in dB) and decoding rates (in frames/s) using the proposed voltage scaling for the different architectures for three different SERs. It can be seen that as the number of cores in the architecture increases from 2 to 4, the power consumption reduces since the operating voltage of the processing cores can be scaled. However, as the number of cores in the architecture increases from 4 to 6, the PSNR reduces because of the increase in the register usage (due to duplication of memory resources among cores, Figure 5.9(b)) and the subsequent increase in the injected SEUs. This makes it harder to scale the processing cores and gives higher power consumption, while maintaining acceptable application-level correctness ($30dB$ PSNR, Section 5.2.4) and real-time performance (Table 5.1).

Allocation	**SER,** λ_0 $(\times 10^{-8})$	*Scaling Coefficients*						P, mW	Ω, dB	D_r, *frames/s*
		Core 1	**Core 2**	**Core 3**	**Core 4**	**Core 5**	**Core 6**			
2 Cores	3.98	1	1	-	-	-	-	14.7	71	33.5
	5.75	1	1	-	-	-	-	14.7	52	33.5
	8.53	1	1	-	-	-	-	14.7	33	33.5
3 Cores	3.98	2	1	2	-	-	-	9	35	30
	5.75	2	1	1	-	-	-	13.6	30	36.5
	8.53	1	1	1	-	-	-	19.6	35	45
4 Cores	3.98	2	3	3	2	-	-	3.6	31	29
	5.75	2	2	2	2	-	-	5.5	30	34
	8.53	1	1	1	1	-	-	19.1	30	55
5 Cores	3.98	2	2	2	2	2	-	5.8	30	38.3
	5.75	2	2	2	1	2	-	13.9	30	49.5
	8.53	3	2	2	1	1	-	18.4	32	54.3
6 Cores	3.98	2	2	2	3	2	3	9.1	31	33.5
	5.75	2	2	2	2	2	1	20.5	31	44
	8.53	2	3	1	1	2	1	37.5	30	54

TABLE 5.12: Power consumption (in mW), PSNRs (in dB) and decoding rates (in frames/s) for different architecture allocations using the proposed power minimisation technique

Similar trade-off between power consumption and application-level correctness was observed with increased SERs. For example, at SER of 5.75×10^{-8} more SEUs are injected in the processing cores (causing degradation of application-level correctness in terms of PSNR) and as such the proposed voltage scaling technique limits the scaling factors on processing cores to maintain acceptable application-level correctness and decoding rates. This results in an increased power consumption of 13.9mW for 5 cores (for example) at SER of 5.75×10^{-8} compared to 5.8mW at SER of 3.98×10^{-8}. Note that depending on the soft error rate, the architecture that gives the minimum power using proposed voltage scaling technique also varies due to the trade-off between application-level correctness and power consumption. For example, architecture with 4 allocated cores give the minimum power among all architectures for SERs 3.98×10^{-8} and 5.75×10^{-8}, while 2 allocated cores give the minimum power among all allocations for SER 8.53×10^{-8}.

5.7 Synthetic Application Examples

In Sections 5.5 and 5.6, MPEG-2 decoder is used as a case study to validate the proposed voltage scaling technique. In this section, more validations of the proposed power minimisation technique is investigated using synthetic application examples. Since the synthetic applications do not describe the nature of processing and output involved, the application-level correctness is hypothetically defined as a direct implication of the overall soft error rate (SER) resulting from voltage scaling on processing cores. Hence, the constraint in (5.20) needs to replaced by an overall SER constraint for synthetic applications given by (5.17). Using (5.17), the overall SER constraint can be expressed as

$$\lambda = \sum_{i=1}^{C} \rho_i \sum_{j=1}^{S} s_{ij} \lambda_{ij} \leq \lambda_{ref} \quad . \tag{5.21}$$

The real-time performance (in seconds) can be found through dividing the multiprocessor execution time (T_M, in clock cycles) by effective multiprocessor frequency due to voltage scaling, f_{eff}, given by (5.13). Hence, the real-time performance constraint is given as

$$\frac{T_M}{\sum_{i=1}^{C} \alpha_i \sum_{j=1}^{S} s_{ij} f_{ij}} \leq T_{M_{ref}}, \tag{5.22}$$

where $T_{M_{ref}}$ is the specified overall real-time constraint (in seconds). The proposed voltage scaling technique can be employed to give minimised power for the synthetic applications through linear programming using the cost function in (5.9) and constraint functions in (5.10), (5.22) and (5.21).

To validate the proposed technique further, a number of synthetic application examples

using random task graphs with 20, 40, 60, 80 and 100 tasks are used. The random task graphs are generated using the random task and resource graph tool [179] (for sample task graphs, see Appendix C). The cost and the number of dependent tasks in the random task graphs are generated using uniform probability distribution with computation cost between 1 and 30, communication cost between 1 to 10 (all costs as multiples of 3.5×10^6 clock cycles), task register usage between 1kbits to 5kbits and the number of dependants was found by exponential distribution between 0 to $N/2$, where N is the number of tasks. The deadline for random task graphs are set to 15, 20, 30, 40 and 50 seconds for random task graph with 20, 40, 60, 80 and 100 tasks, respectively. Also, the SER constraint in (5.21) is set to an arbitrary SER of λ_{ref}=10^{-7} for illustration purposes. Table 5.13 shows the power consumption (in mW) and the number of SEUs experienced by the random task graphs for different architecture allocations using mapping M4. The fault injections were carried out using a base soft error rate (SER) of 3.98×10^{-8}, while the power values were found using (5.1). The applications are shown in column 1, while the power consumption (P) and the overall SER (λ) of each architecture allocation (from 2 cores to 6 cores) are shown in columns 2-6 (Table 5.13).

Application	2 Cores		3 Cores		4 Cores		5 Cores		6 Cores	
	P, mW	λ ($\leq\lambda_{ref}$)	P, mW	λ ($\leq\lambda_{ref}$)	P, mW	λ ($\leq\lambda_{ref}$)	P, mW	λ ($\leq\lambda_{ref}$)	P, mW	λ ($\leq\lambda_{ref}$)
20 tasks	10.2	7.41E-8	4.9	9.95E-8	7.3	9.95E-8	7.3	9.91E-8	18.8	8.17E-8
40 tasks	10.2	7.23E-8	8.5	8.92E-8	8.9	9.86E-8	7.0	9.95E-8	22.9	7.28E-8
60 tasks	11.5	6.70E-8	9.6	8.46E-8	9.3	9.90E-8	6.6	9.92E-8	16.1	7.95E-8
80 tasks	11.2	7.23E-8	8.9	9.54E-8	10.1	9.90E-8	7.0	9.89E-8	15.5	8.79E-8
100 tasks	18.0	3.98E-8	11.4	7.79E-8	11.4	8.87E-8	15.2	9.82E-8	19.7	7.46E-8

TABLE 5.13: Power consumption (P, in mW) and the overall SERs (λ) for different synthetic applications for different architecture allocations

From Table 5.13 the following two observations can be made. First observation is related to the trade-off between power minimisation and reliability in terms effective multiprocessor SER due to voltage scaling. For example, lower overall SER (λ) of 6.70×10^{-8} for task graph with 60 tasks is achieved at high power consumption (P) of 11.5mW using architecture allocation of 2 processing cores. On the other hand, higher overall SER (λ) of 9.92×10^{-8} for the same task graph is achieved at lower power consumption (P) of 6.6mW. Second observation is related to the fact that the power consumptions (in mW) for different architecture allocations vary depending on the given deadline and SER constraints (given by (5.22) and (5.21)). For example, for the given real-time constraint of 20 seconds and SER constraint of 10^{-7}, the task graph with 40 tasks gives the minimum power of 8.5mW for architecture with 3 cores, while it gives the maximum power of 22.9mW for architecture with 6 processing cores. The higher power consumption for architecture allocation with higher number of cores is due to higher register usage (Section 5.6.2). Such higher register usage limits voltage scaling on cores as overall SER increases, given by (5.21) leading to high power consumption.

5.8 Concluding Remarks

Resilience against SEUs and low power consumption are key objectives in the design of emerging MPSoCs. However, these are conflicting objectives as low power design techniques, such as DVS, exacerbate the soft error rate. Recently the concept of application-level correctness was introduced showing the impact of SEUs at application-level rather than architectural-level. This chapter has presented the first investigation that considers the relationship between application-level correctness and supply voltage. The relationship between soft error rate and application-level correctness has been established through exhaustive statistical analysis and curve-fitting technique. Based on this relationship, a novel soft error-aware power minimisation technique has been proposed formulating the optimisation as a linear programming problem with an aim to give optimised voltage level for each processing core. Using MPEG-2 video decoder and other synthetic examples, significant power reduction has been shown using the proposed technique. Furthermore, the effect of MPSoC architecture allocation and application task mapping on the trade-off between application-level correctness and power minimisation has been examined.

Chapter 6

Soft Error-Aware Design Optimisation

Chapter 5 presented a voltage scaling technique for power minimisation of MPSoCs. It was shown that significant power reduction can be achieved, while maintaining acceptable reliability at application-level and meeting a specified real-time performance. This chapter examines the impact of application task mapping on the reliability of MPSoC in the presence of single-event upsets (SEUs). A novel soft error-aware design optimisation is presented using joint power minimisation using voltage scaling and reliability improvement through application task mapping. The aim is to minimise the number of SEUs experienced by the MPSoC for a suitably identified voltage scaling of the system processing cores such that the power is reduced and the real-time constraint is met. The effectiveness of the proposed design optimisation technique is demonstrated using different applications, including an MPEG-2 video decoder and random task graphs. Furthermore, the impact of architecture allocation (varying the number of MPSoC cores) is investigated on the power consumption and SEUs experienced using the proposed design optimisation technique.

The rest of this chapter is organised as follows. Section 6.1 presents review of related works. Section 6.2 presents the system model and Section 6.3 demonstrates the impact of application task mapping on reliability. Section 6.4 presents the novel design optimisation technique and Section 6.5 shows the experimental results and compares with soft error-unaware design optimisation techniques. Section 6.6 investigates the effect of architecture allocation (choice of number of processing cores) on the proposed design optimisation technique. Finally, Section 6.7 concludes the chapter.

6.1 Related Works

Dynamic voltage scaling (DVS) is an effective power minimisation technique often employed in hand-held devices to extend the battery life [66]. The DVS technique works by lowering the processor voltage and frequency according to its workload to achieve power reduction [48] (a brief introduction to DVS technique is presented in Section 2.2, Chapter 2). However, it has been reported that the reduction of supply voltage causes an exponential increase in the rate of soft errors, particularly that of single-event upsets (SEUs), leading to degradation of reliability [88, 148]. This is further exacerbated by device miniaturisation and continuing technology scaling [14]. As a result, reliability is emerging as a key challenge for low power system design [63, 88, 89, 91, 148, 149].

Several researchers have proposed number of power-aware fault tolerance or soft error hardening techniques to mitigate the effect of increased soft errors caused by power minimisation. Redundancy is a popular fault tolerance technique. This technique is generally effective in that the redundant resources provide a voting system from multiple resources to produce a single output in the presence of soft errors or faults [180]. Over the years, a number of redundancy-based techniques have been reported at various levels of design abstraction. The triple modular redundancy (TMR) is the most basic circuit-level redundancy technique, such as [180, 181]. The fault tolerance is achieved through TMR technique using three hardware elements to incorporate voting a single output from multiple outputs. Due to the increase in hardware resources, large overhead in terms of power consumption and chip area is incurred using such technique [182, 92]. Another effective technique in terms of power consumption is the time redundancy, such as [93, 94]. Such technique employs multiple instances of execution to achieve fault tolerance. Although no overhead in terms of hardware resources or area is caused, this technique has overhead in terms of performance. Information redundancy proposed in [183, 184] is also an effective fault tolerance technique. The main idea is to append error-detection and error-correction codes along with the usual information bits to increase reliability of the system [183]. The addition of these extra codes add to the communication and computation overhead, while improving the reliability in the presence of soft errors [123]. Recently joint time and information redundancy-based technique has been proposed in a number of publications, such as [62, 185]. Using such technique has advantages of high reliability or fault tolerance at low cost and low overhead.

Alternatives to the redundancy-based technique are the re-execution and replication of computational tasks among idle processing elements. Using this technique, no overhead in incurred in terms of extra execution time. For example, in [13, 186] low power fault tolerance technique has been presented utilising the idle processing elements for duplicating some of the computations. Other flexible techniques to achieve high fault tolerance are the pre-emptive on-line scheduling, such as [187] and check-pointing dur-

ing slack times between tasks, such as [97]. Using these techniques, high fault tolerance can be achieved at the cost of increased complexity in the design of MPSoC application. However, the effectiveness of these techniques depends upon predictability of slack times, which incur large overheads and often leads to problems related to unschedulability [74]. Also, achieving fault tolerance using check-pointing technique is limited by the schedulability and the number of check-points. The optimal number of check-points that can be inserted and scheduled in the presence of a given number faults is determined by the worst case execution time of a task [188].

Recently, researchers have shown combination of different fault tolerance techniques to reduce system overhead for fault-tolerant and low power design. For example, fault tolerance-based optimisation of cost-constrained distributed real-time systems has been proposed in [74]. The fault tolerance in [74] is achieved through mapping and assignment of different fault tolerance policies to processes. Another fault-tolerant design using process re-execution and re-scheduling of low power heterogeneous MPSoC applications has been proposed in [99]. The power minimisation is achieved through scheduling of voltage levels to different processes and the fault tolerance is achieved by deciding the start times of processes and the transmission times of messages in the presence of faults. In [50] a dynamic fault tolerance technique is presented using independent task sets scheduling with precedence relationship in MPSoC systems. Due to the use of such scheduling, the fault tolerance technique in [50] benefits from less communicational complexity and better scheduling performance in terms of power consumption.

Traditionally power-aware fault-tolerant design techniques consider low power and reliability as two separate objectives [13, 74, 187]. For effective design optimisation with low power and improved reliability as a joint objective, further studies are needed to understand reliability of applications, particularly from system- and application-level design perspective. Application task mapping is one such crucial system-level design step of the MPSoCs (for introduction to different system-level design steps, see Section 2.4.2.2, Chapter 2). A number of studies have been reported showing the impact of the application task mapping on the system performance [189] and system energy consumption [190]. However, currently no study exists that examines the impact of the application task mapping on the reliability of an application. This chapter presents the first study of the impact of application task mapping on the reliability of MPSoC in the presence of SEUs.

6.2 System Model

In this section, the models of the system architecture, application and fault injection used in this work are introduced.

6.2.1 Architecture Model

In this work, an MPSoC architecture, A, based on 2D-mesh network-on-chip (NoC) with C processing cores is considered. Due to its high performance [6] (see Section 4.4, Chapter 2 for comparative analysis) and scalability, such NoC-based MPSoC architectures are gaining popularity and a number of academic or industrial designs have been proposed to date, such as xPIPES [87], Intel 80-core [125]. Figure 6.1 shows an example MPSoC architecture consisting of four processing cores. As shown, each block (or NoC tile) consists of a processing core and switch (Figure 6.1). Each processing core consists

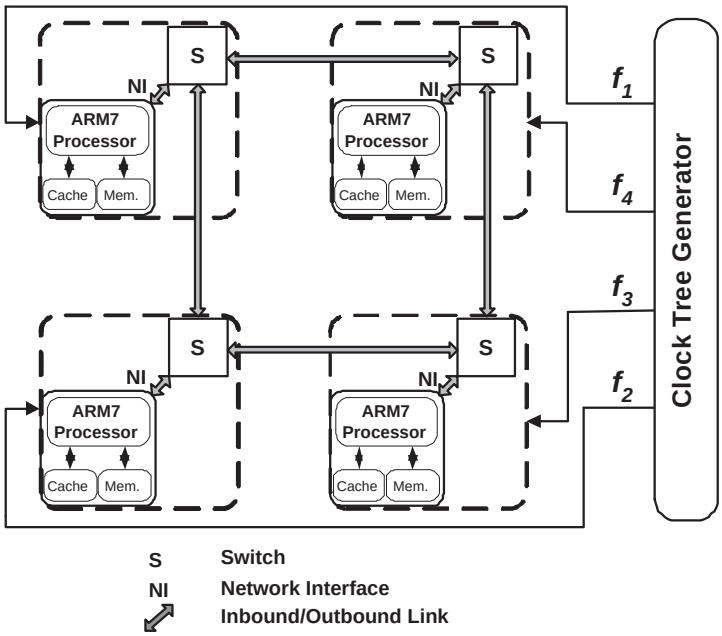


FIGURE 6.1: MPSoC architecture with four processing cores and power minimisation support through clock tree generator

of, among others, ARM7TDMI processor, an instruction cache (8kbits), a data cache (16kbits) and a dedicated private memory (256kbits). Network interface attached to each processor (Figure 6.1) incorporates packet-based communication with 32-bit payload and switches carry out inter-core packet-based communication with XY routing, chosen due to its performance and simplicity [6]. The cache and memory sizes have been chosen to provide high availability of data and parallelism among the processing cores. To introduce power minimisation through voltage scaling in the MPSoC architecture, a clock tree generator has been included to provide different clock frequencies for the processing cores through voltage scaling (Figure 6.1). Similar voltage scaling arrangement was also used in Chapter 5 (Section 5.2.2). For a given voltage scaling, the dynamic power, P, of a processing core is given as

$$P = \alpha C_L f {V_{dd}}^2 \quad , \tag{6.1}$$

Scaling, s	f**, MHz**	V_{dd}**, V**
1	200	1.00
2	100	0.58
3	66.7	0.44

TABLE 6.1: Operating frequency, f, and supply voltage, V_{dd}, for different voltage scaling of ARM7TDMI processor

where C_L is the processor load capacitance per cycle, V_{dd} is the supply voltage, f is the operating frequency and α is the processor activity factor ($0 \leq \alpha \leq 1$). The voltage scaling technique effectively reduces the power consumption, defined by (6.1), by reducing V_{dd} and f through scaling. For ARM7TDMI processor, the empirical relationship between V_{dd} (in volts) and f (in MHz) is given by [47] as

$$V_{dd}(f, s) = \left[0.1667 + \frac{4.1667 \times f}{10^3 \times s}\right], \tag{6.2}$$

where s is the frequency scaling constant. Table 6.1 shows the three voltage scaling options (with s=1, 2 and 3 in (6.2)) used in this work. The impact of choice of voltage scaling levels on design optimisation is discussed in Section 6.5.

6.2.2 Application Model

An application is modelled as a directed, acyclic task graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with N nodes. Each node $t_j \in \mathcal{V}$ represents j-th computational task within the application and each edge $d_{j,k} \in \mathcal{E}$ represents inter-task communication and data dependency between j-th and k-th tasks (j, k=1:N, where N is the number of computational tasks in the task graph). An application is realised on the MPSoC architecture by distributing the computation and communication tasks among the processing cores and their interconnects through application task mapping. Figure 6.2 shows an example task graph of MPEG-2 video decoder using eleven tasks and their associated register resources (brief introduction to MPEG-2 video decoder is given in Appendix A). The computational and communication costs of tasks are shown with numbers on the nodes and edges, Figure 6.2. The computational cost represents execution time of each task and the communication cost represents the time required to transfer data between tasks (actual costs are approximate multiples of 5.5×10^6 clock cycles). The computational and communication costs are obtained using SystemC cycle-accurate simulations assuming 32-bit inter-core transfer. The computational tasks are modelled as separate task processes, while the communication between tasks is modelled as message passing queues. The communication cost is found by dividing the size of inter-task queue by the bandwidth of the channel (in bits per cycle). Similar evaluation of computation and communication costs has also been used in [74, 172].

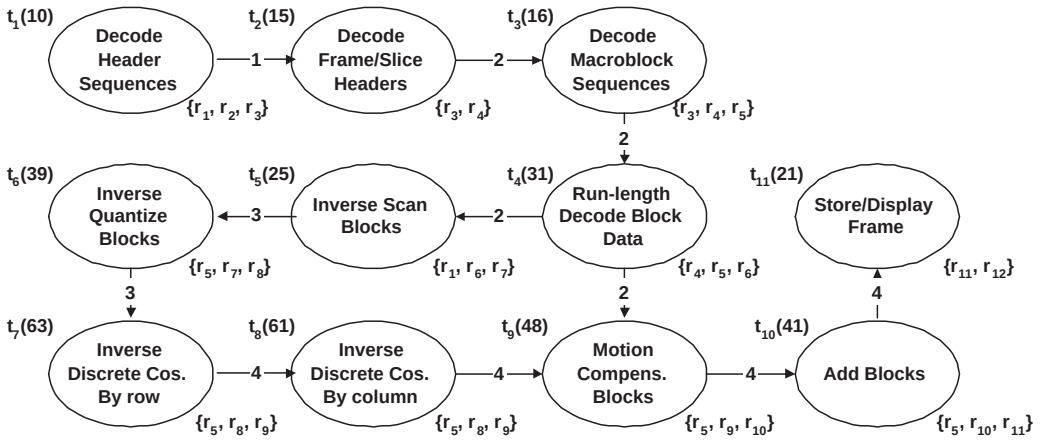


FIGURE 6.2: MPEG-2 video decoder task graph with eleven tasks and associated register resources

Register set	**Type**	**Size,** *bits/cyc.*
r_1	Scan tables and variables	4132
r_2	Sequence types	1124
r_3	Header sequence variables (before decoding)	640
r_4	VLC tables and variables	5124
r_5	Header sequence variables (after decoding)	2134
r_6	Video blocks (coded)	12288
r_7	Scanned video blocks	13218
r_8	Quantised video blocks	13132
r_9	Picture ready video blocks (before motion compensation)	13274
r_{10}	Motion compensated video data and variables	13326
r_{11}	Decoded and motion compensated video data	13174
r_{12}	display/storage ready video data structure	12288

TABLE 6.2: Register usage of MPEG-2 video decoder tasks (Figure 6.2) and their approximate sizes

Also attached with each node is a set of application registers showing the register usage by the computational tasks. Due to inter-dependent nature of the tasks of an application, the tasks share register resources among themselves. For example, the task t_1 uses the set of registers r_1, r_2 and r_3, while the task t_2 uses the set of registers r_3 and r_4. Note that between these two tasks, r_3 is shared. Table 6.2 shows the different register sets used by the MPEG-2 video decoder tasks along with their types and approximate sizes (obtained by using variable or signal type tags within the SystemC simulation environment). The actual register usage of the i-th processing core ($i = 1 : \mathcal{C}$, where $\mathcal{C}$ is the number of processing cores of an MPSoC), R_i, is found through SystemC simulation after the

application tasks and their associated registers (Table 6.2) are mapped on the processing cores of an MPSoC. The R_i is given as

$$R_i = \frac{1}{T_i} \sum_{t=1}^{T_i} R_{i,t} \quad , \tag{6.3}$$

where $R_{i,t}$ is the register usage (in bits) at t-th clock cycle of the i-th processing cores. The R_{i_t} in (6.3) depends on the number of tasks mapped with associated resources (Table 6.2).

6.2.3 Fault Injection Model

In this work, fault injection is carried out using SEU-based fault model employing the fault injection simulator proposed in Chapter 3. Using this technique the injection of SEUs is initiated through replacement of variable or signal types in the original design specification to equivalent fault injection enabler types. These fault injection enabler types help form fault locations database for the device under test, which contains the target registers for SEU injection. The fault injection simulator injects SEUs in these target registers based on the specified soft error rates and probability distribution for determining fault locations. Figure 6.3 shows the fault injection setup used for the MPSoC architecture with four processing cores as an example. As can be seen, four fault locations databases are formed for four processing cores through replacement of variable/signal types to fault injection enabler types. For a given base soft error rate (SER, λ_0, in SEUs per bit per cycle considering no scaling), the actual SERs (λ, in SEUs per bit per cycle after scaling) for processing cores are found by the corresponding voltage settings used. For these actual SERs, the SEUs are injected at random locations determined by Poisson's distribution within the register space of the fault locations database. To control fault injection timings the system clock is connected to the fault injection simulator. Using SystemC monitor modules in cycle-accurate simulations, register usage and the number of SEUs experienced are found (Figure 6.3).

6.3 Impact of Task Mapping on Reliability

Reliability of an MPSoC application in the presence of SEUs is related to the total number of SEUs experienced [24]. For a soft error rate (SER) of λ_i (SEUs per bit per clock cycle), the total number of SEUs experienced, Γ, by an MPSoC with $\mathcal{C}$ processing cores is given by (3.3) (Section 3.2.3, Chapter 3) as

$$\Gamma = \sum_{i=1}^{\mathcal{C}} R_i T_i \lambda_i = T_M \sum_{i=1}^{\mathcal{C}} R_i \alpha_i \lambda_i \quad , \tag{6.4}$$

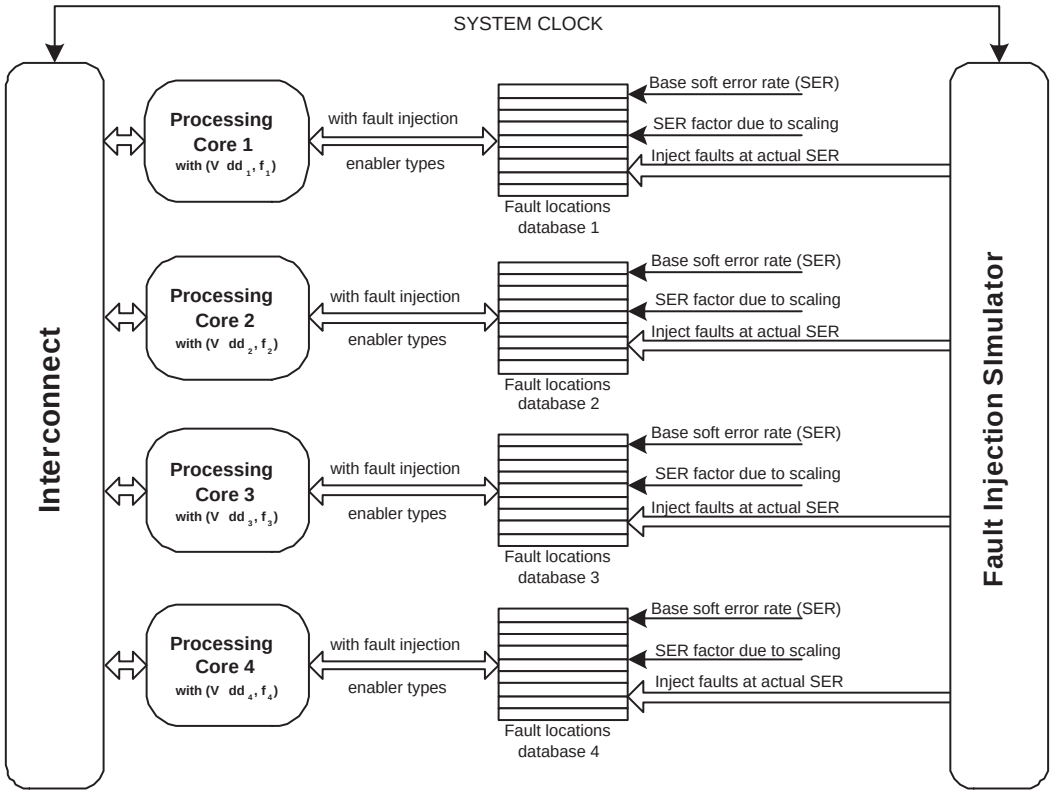


FIGURE 6.3: Fault injection setup for MPSoC architecture with four processing cores

where T_i is the execution time, R_i is the register usage of i-th processing core and T_M is the multiprocessor execution time (in clock cycles, $\forall_i$:T_i=$\alpha_i T_M$) of the i-th processing core. The register usage of a processing core, defined by (6.3), depends on the nature of processing being carried out by the tasks mapped, data dependency and resource sharing among them. The T_M in (6.4) affects multiprocessor performance and depends on the number of mapped tasks on a processing core and the data dependency among them. As a result, when more related tasks are mapped on a processing core, T_M increases for a given operating frequency but the register resources related to tasks are localised reducing the overall register usage (R=$\sum_i R_i$). On the other hand, when tasks are distributed among processing cores to achieve higher parallelism, T_M decreases at the expense of increased R due to higher duplication of shared register resources among tasks. Examples of this trade-off follow. In the MPEG-2 decoder (Figure 6.2), the tasks t_5 and t_6 share about 13kb registers (r_7 is shared between them), while the tasks t_6, t_7 and t_8 share approximately 15kb registers among them (r_5 and t_8 are shared among them). To reduce register usage, for example it is possible to map tasks t_5, t_6, t_7 and t_8 on a processing core through localisation of the registers. However, due to computationally intensive nature of these tasks, T_M will be high (with total computation cost of 188). To reduce T_M, an alternative option is to map tasks t_5 and t_6 on a processing core, while the tasks t_7 and t_8 can be mapped on another core. However, this gives a duplication of about 15kb registers (increased register usage) between the processing cores. Because

of this register usage (R) and multiprocessor execution time (T_M) trade-off, the MPSoC experiences varying number of SEUs (Γ) for different task mappings given by (6.4). The Γ also depends on the voltage scaling of the MPSoC processing cores as it affects λ_i in (6.4). To demonstrate the impact of application task mapping and voltage scaling on the number of SEUs experienced (Γ), a total of 120 random task mappings were carried out using the MPEG-2 decoder (Figure 6.2) on the MPSoC architecture (Figure 6.1). Figure 6.4 shows the T_M, R and Γ obtained through SystemC simulation and fault injection (Section 6.2.3) using an SER of 10^{-9} SEU per bit per cycle (i.e. 1 SEU per 10ms for 1kb register bank) as an example. Three key observations are made:

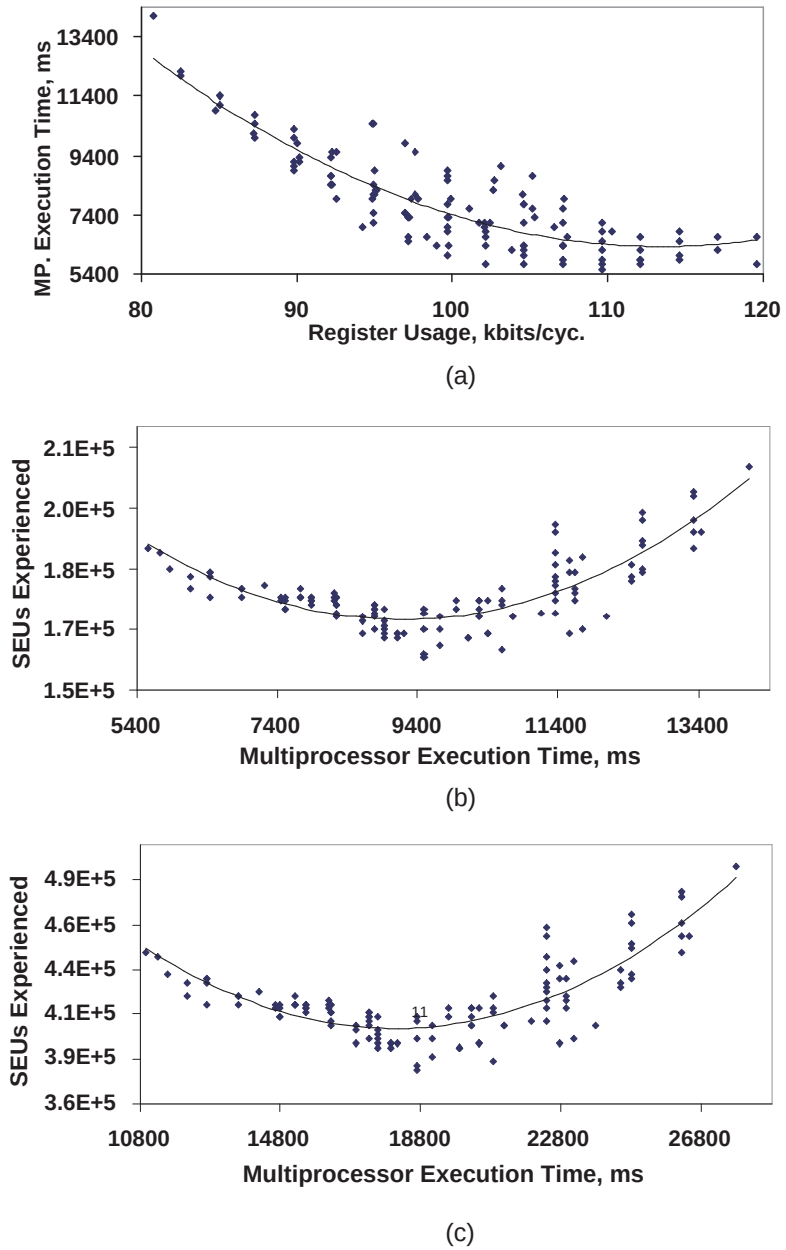


FIGURE 6.4: (a) Trade-off between multiprocessor execution time (in ms) and register usage (in kbits/cycle), (b) SEUs experienced and multiprocessor execution time (in ms) when no scaling is used for MPSoC cores, and (c) SEUs experienced and execution time when MPSoC cores are scaled by 2; all for different task mappings of MPEG decoder with four processing cores

Observation 1 : Figure 6.4(a) shows the trade-off between multiprocessor execution time (T_M, ms) and overall register usage (R). As can be seen, when tasks are mapped to reduce R by localisation of tasks, T_M increases. On the other hand, as tasks are mapped to reduce T_M, register resources shared among tasks are duplicated, leading to increased register usage, R.

Observation 2 : Figure 6.4(b) shows the total number of SEUs experienced (Γ) and multiprocessor execution time, T_M (in ms), when all the decoder processing cores are scaled by 1 (f=200MHz and V_{dd}=1V). It can be seen that when tasks are distributed among processing cores to reduce T_M, the decoder experiences more higher Γ given by (6.4) due to higher R (Figure 6.4(a)). When tasks are localised to reduce R, the decoder also experiences higher number of SEUs due to increased T_M (Figure 6.4(a)). This results in a concave curve for Γ given by (6.4), with the minimum Γ located around the middle of T_M range.

Observation 3 : Figure 6.4(c) shows the total number of SEUs experienced (Γ) and multiprocessor execution time (T_M, in ms) when all the decoder processing cores are scaled by 2 (f=100MHz and V_{dd}=0.58V). As can be seen, Γ increases by approximately 2.5 times due to V_{dd} scaling from 1V to 0.58V (found through V_{dd} and λ relationship shown in [170], see Section 5.3, Chapter 5 for further details) and T_M is increased by a factor of 2 due to reduced f from 200MHz to 100MHz. For example, for an application task mapping Γ increases from 1.71×10^5 to 4.12×10^5, while T_M increases from 9.5s to 18.2s due to scaling of V_{dd} from 1V to 0.58V.

The above observations demonstrate the impact of application task mapping and voltage scaling on the MPSoC decoder reliability in the presence of SEUs. Hence, an interesting design optimisation problem is to identify suitable voltage scaling of the MPSoC processing cores to minimise power consumption and to improve reliability through application task mapping, while meeting a real-time constraint.

6.4 Proposed Design Optimisation

To solve the problem of joint power minimisation and reliability improvement of an application, a novel design optimisation is proposed. Figure 6.5 shows flowchart of the proposed design optimisation with three major steps: power minimisation (step 1), soft error-aware application task mapping (step 2) and iterative assessment (step 3).

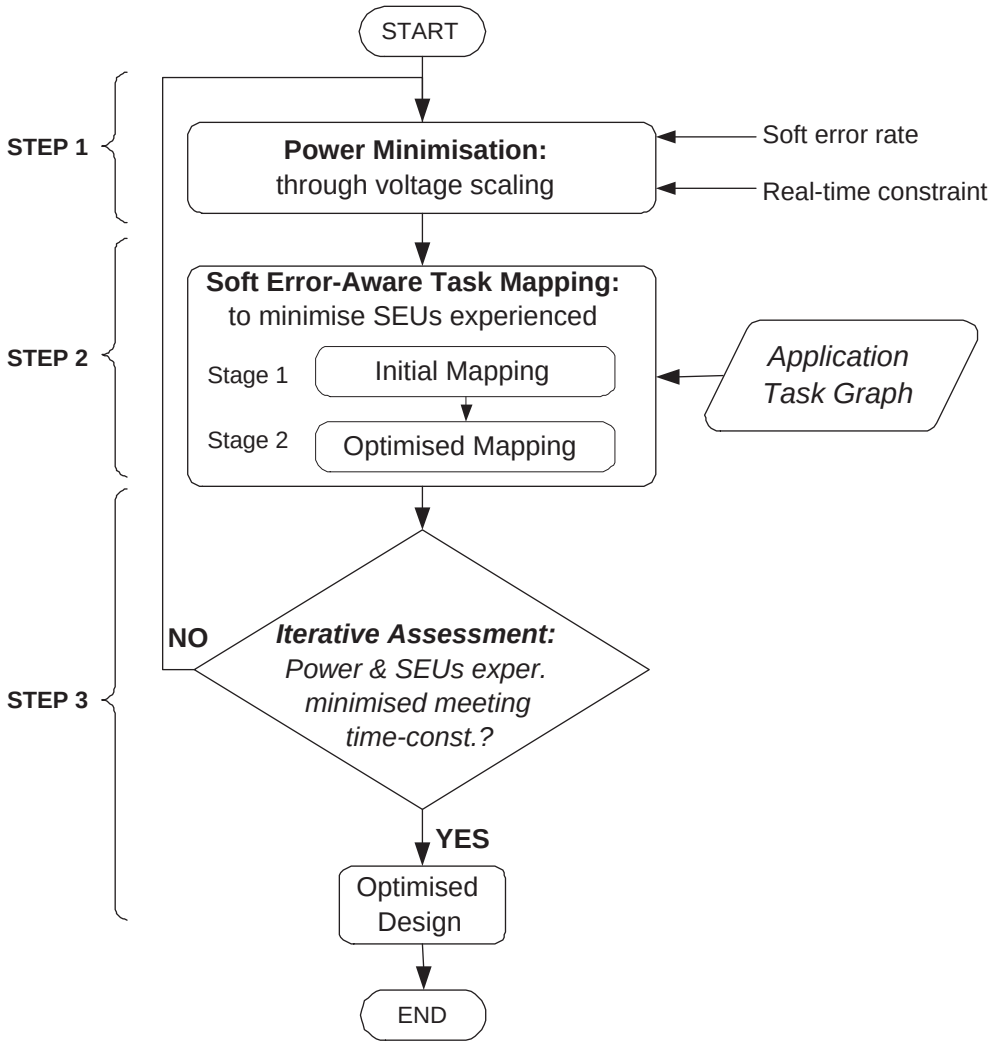


FIGURE 6.5: Flowchart of the proposed design optimisation

For a given SER and real-time constraint, the design optimisation is initiated by power minimisation (step 1) through voltage scaling of the MPSoC cores. This is followed by soft error-aware application task mapping (step 2) to minimise the number of SEUs experienced for the chosen voltage scalings in step 1. These two steps are repeated and assessed in step 3 to find a design with minimised power consumption and improved reliability in terms of SEUs experienced, while meeting the real-time constraint. The design optimisation steps are discussed next.

6.4.1 Power Minimisation

Power minimisation in the proposed design optimisation is performed using the voltage scaling algorithm, Figure 6.6(a). The voltage scaling algorithm, *nextScaling*, starts with the lowest voltage scaling on all identical cores and generates the next set of higher voltage scaling, *nextS*, based on the previous set of coefficients, *prevS* (Figure 6.6(a)). In each iteration, *nextS* is updated as the *prevS* reduced by 1 on a processing core until the voltage scaling on the core reaches the nominal voltage scaling level (s=1, lines 3-6). When the nominal level (s=1) is reached, *nextS* of the core is updated by increasing voltage scaling of the core by 1 (line 9) and reducing the voltage scaling of the next processing core by 1 in steps (lines 7-11). The aim is to generate non-repetitive combinations and reduce the number of voltage scalings that need to be investigated. For example, for a homogeneous architecture with four processing cores (Figure 6.1) and three scaling options (Table 6.1), the voltage scaling scaling algorithm, Figure 6.6(a), generates 15 unique combinations, Figure 6.5(b), compared to a total of 3^4=81 possible combinations. As can be seen, the voltage scaling starts with the highest scaling coefficient of 3 for all cores, followed by 3 for core 1, core 2, core 3, and 2 for core 4 as the next scaling combination (Figure 6.6(b)). As core 4 reaches nominal value of 1, the next combination is generated by *nextScaling* algorithm as 3 for core 1, core 2, and 2 for core 3 and core 4. This is followed by 3 for core 1, core 2, 2 for core 3, and 1 for core 4. The voltage scaling algorithm, thus, effectively generates all possible combinations. For a set of scaling coefficients from voltage scaling algorithm (Figure 6.6(a)), the dynamic power consumption, P, of the MPSoC with $\mathcal{C}$ processing cores can be expressed as a function of voltages scaling coefficient, s_i, as

$$P = C_L \sum_{i=1}^{\mathcal{C}} \alpha_i f_i(s_i) V_{dd_i}^2(s_i), \tag{6.5}$$

where $f_i(s_i)$ and $V_{dd_i}(s_i)$ take values depending on the voltage scaling coefficient s_i (Table 6.1). For each set of voltage scaling coefficients, soft error-aware application task mapping is carried out (step 2, Figure 6.5) to minimise the number of SEUs experienced.

```
//C = no of cores, prevS = previous scaling
[nextS] = nextScaling(prevS): begin
 1: copy prevS into nextS
 2: for i := 1 to C
 3:   if prevS[i] > 1: begin //1 is lowest scale
 4:     nextS[i] := prevS[i]-1;
 5:     break;
 6:   end if
 7:   else
 8:     for k := i to C
 9:       nextS[k] = prevS[k]+1;
10:     end for
11:   end else
12: end for
13: return nextS;
end
```

(a)

	Scaling Coefficients			
Iter. #	Core 1, s_1	Core 2, s_2	Core 3, s_3	Core 4, s_4
1	3	3	3	3
2	3	3	3	2
3	3	3	3	1
4	3	3	2	2
5	3	3	2	1
6	3	3	1	1
7	3	2	2	2
8	3	2	2	1
9	3	2	1	1
10	3	1	1	1
11	2	2	2	2
12	2	2	2	1
13	2	2	1	1
14	2	1	1	1
15	1	1	1	1

(b)

FIGURE 6.6: (a) Voltage scaling algorithm used for power minimisation, (b) example of voltage scaling coefficients for four processing cores using voltage scaling algorithm shown in (a)

6.4.2 Soft Error-Aware Application Task Mapping

The problem of application task mapping on MPSoC cores to minimise SEUs experienced (Γ) is an NP-complete problem [74]. In this section, a soft error-aware application task mapping is developed in two stages (step 2, Figure 6.5): the stage 1 is the initial soft error-aware application task mapping, followed by stage 2 of search-based optimised application task mapping. Figure 6.7 shows the initial soft error-aware application task mapping algorithm (stage 1), *InitialSEAMapping*, which aims to simplify the optimisation process by reducing the number of task movements. The *InitialSEAMapping* starts with mapping the task with no predecessor in task graph (G) (line 1). The dependants of the currently mapped task in G are then sorted by SEUs experienced if they are to be mapped with the current task. The sorted list of dependants is stored in L (line 5). The task that gives the minimum SEUs in L is then mapped next (lines 6-10). This is continued until the execution time of the current core does not exceed the real-time constraint ($T_{M_{ref}}$) and the number of unmapped tasks left in task graph G is higher than the number of remaining cores to ensure that tasks are mapped in all cores (lines 4-13). The unmapped tasks are stored in a queue, Q (line 10), which are then mapped gradually to the other cores using the same criteria. After all tasks are mapped, the initial mapping (M) is returned by *InitialSEAMapping* (lines 6-15).

After the initial soft error-aware task mapping (*InitialSEAMapping*, Figure 6.7), the design optimisation is continued further through optimised mapping (stage 2, step 2, Figure 6.5). The optimisation is carried out through iterative search-based mapping algorithm, *OptimisedMapping*, Figure 6.8, employing the list scheduling for mapped tasks.

```
//  C: no of cores,  G: application task graph with      N tasks, t is the current task
//M: mapping of all cores,       A : MPSoC arch.,  Q: task queue,  L: temporary list
[M] = InitialSEAMapping  (G, C, A):  begin
 1:  push G[0] into Q     // push the first task into Q
 2:  for i:= 1 to C-1  and  Q is not empty
 3:    t := Q.front(); M[i].map(t); delete all mapped tasks from Q
 4:    while  Ti < T Mref  and  no. of unmapped tasks in G > (C-i)
 5:      L : = dependents of    t //(sorted by minimum SEUs)
 6:      if L is empty  and  Q is not empty
 7:        swap last two elements in Q
 8:      else if  Q is not empty
 9:        t = first element in L    //task with minimum SEUs and Time
10:      M[i].map(t); delete t from L; move tasks in L into Q and empty L
11:    else  break  while; end if
12:    t = Q.front();
13:  end while
14: end for
15: return  M;
end
```

FIGURE 6.7: Initial soft error-aware mapping algorithm, *InitialSEAMapping*

The aim of such scheduling is to make an ordered grouping of tasks in processing cores to accommodate different constraints and task dependencies [74]. The *OptimisedMapping*

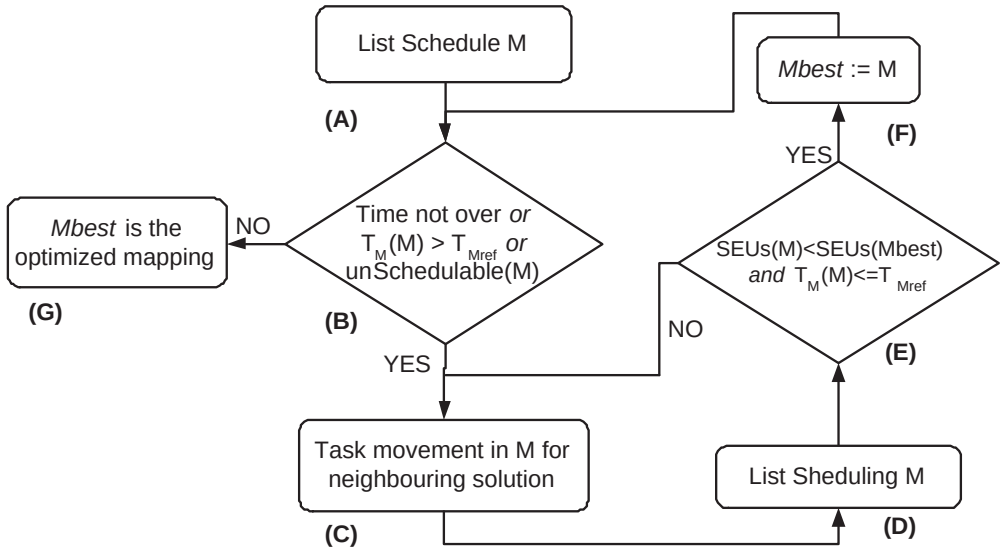


FIGURE 6.8: Flowchart of optimised mapping, *OptimisedMapping*

starts with scheduling the initial task mapping, M (step A, Figure 6.8). The mapping M is then checked to see if it violates the schedulability requirements or real-time constraints (step B). If any such violation is found within the search time, the optimisation proceeds with generating neighbouring task movements to find out a possible next mapping solution (step C). With neighbouring task movement, the new mapping (M) is then list scheduled, if schedulable (step D). This is followed by comparison with the

the previous best solution, *Mbest*. If M is better than *Mbest* in terms of lower number of SEUs experienced and meets the given real-time constraint, it is then updated as the new *Mbest* (steps E-F). The optimisation steps C-F are repeated until the specified search time is not over (step B). Once the search time is over, *Mbest* is returned as the optimised design for the chosen voltage scalings (step G).

In *OptimisedMapping*, the multiprocessor execution time (T_M, in seconds) for an application task mapping is found by the dividing the total number of execution cycles of all mapped tasks by the effective number of cycles executed by processing cores per second for chosen voltage scaling (step B, E, Figure 6.8), i.e.

$$T_M = \left[\sum_{i=1}^{\mathcal{C}}\sum_{j=1}^{N}\left(t_j^i + \sum_{k=1}^{N} d_{j,k}^i\right)\right] / \left[\sum_{i=1}^{\mathcal{C}} \alpha_i f_i(s_i)\right], \tag{6.6}$$

where t_j^i is the execution time (in clock cycles) of the j-th task mapped on i-th processing core, $d_{j,k}^i$ is the dependency cost (in clock cycles) between j-th and k-th task $(j,k = 1:N)$ due to selection of j-th task on i-th processing core. The total number of SEUs experienced (Γ) for an application task mapping with chosen voltage scaling on MPSoC processing cores is found in *InitialSEAMapping* (line 5, Figure 6.7) and *OptimisedMapping* (step E, Figure 6.8) through (6.4). The per core execution time (T_i, in clock cycles) and register usage (R_i, in bits per cycle) in (6.4) are given in terms of mapped tasks as

$$\forall_i : T_i = \sum_{j=1}^{N}\left(t_j^i + \sum_{k=1}^{N} d_{j,k}^i\right), \text{ and} \tag{6.7}$$

$$\forall_i : R_i = Avg\left\{\left|\left(\bigcup_{j=1}^{N}\bigcup_{k=1}^{N} r_{j,k}^i\right)\right|\right\}, \tag{6.8}$$

where $r_{j,k}^i$ is the set of registers shared between j-th and k-th tasks for being mapped on i-th processing core (j=k defines the local register usage of j-th task). As can be seen in (6.8), R_i is given by average cardinality of the register set over the execution time arising out of union of register usages of mapped tasks ($r_{j,k}^i$) in i-th processing core. The proposed optimisation is carried out by iterative search through N tasks, with each iteration generating maximum two task movements out of maximum search space of (N-1) dependent tasks. This is followed by second stage search through maximum (N-1) tasks for minimum number of SEUs experienced. As a result, *OptimisedMapping* has worst-case complexity of $O(2N(N-1)(N-1)) \approx O(N^3)$.

An example illustrating the proposed soft error-aware application task mapping algorithm is shown in Figure 6.9. In Figure 6.9(a), an application task graph with six tasks is shown (all costs are multiples of 60×10^4 cycles) and in Figure 6.9(b)-(c) the application registers and their distribution for different tasks are shown. Figure 6.9(d)-(g)

show the incremental task mapping using *InitialSEAMapping* algorithm, Figure 6.7, and finally, Figure 6.9(h)-(i) show scheduling and task movements using *OptimisedMapping*, Figure 6.8.

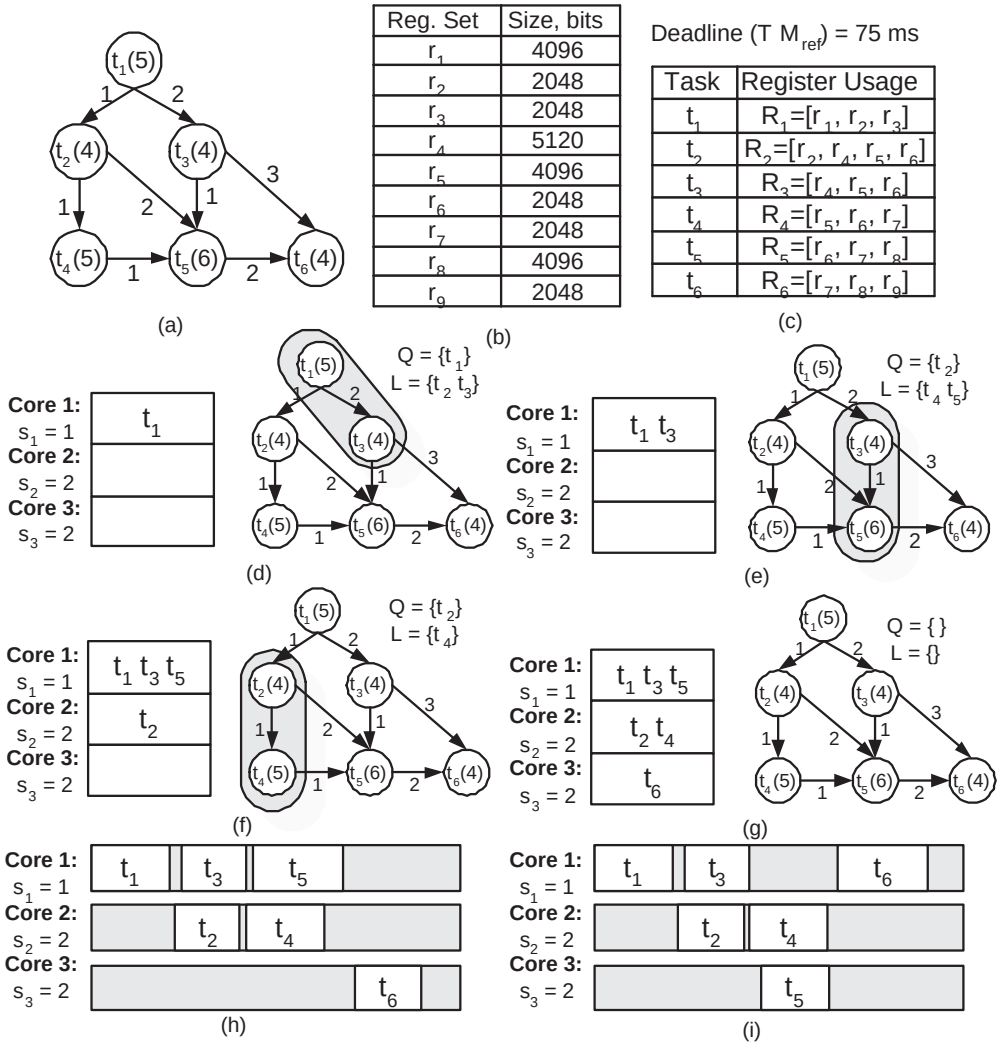


FIGURE 6.9: Example illustration of the soft error-aware application task mapping (a) example application task graph, (b) sets of registers and their sizes, (c) register usage of different tasks of the application, (d-f) initial soft error-aware application task mapping (*InitialSEAMapping*, Figure 6.7) steps, and (g) optimised mapping (*OptimisedMapping*, Figure 6.8) step

The chosen voltage scaling for the processing cores are: s_1=1, s_2=2 and s_3=2 and deadline is assumed to be $T_{M_{ref}}$=75ms. As can be seen, after the first task, t_1, in the application task graph, Figure 6.9(a), is mapped to processing core 1, the *InitialSEAMapping* mapping algorithm selects t_3, followed by t_6 from dependency list, L. This is because task t_3 gives the least number of SEUs experienced compared to t_2 and t_5 shown in gray, Figure 6.9(d), with the $r_{j,k}$ values from Figure 6.9(c). Note that after allocating t_1, t_3 and t_5 on core 1, the deadline constraint cannot be satisfied with further allocation of tasks and the mapping algorithm carries on with mapping of tasks t_2 and t_4 in core 2, which give minimum SEUs experienced, Figure 6.9(f). Finally, the unmapped task t_6 in queue (Q) is mapped to core 3, Figure 6.9(g). After *InitialSEAMapping* (Figure 6.7) is completed, *OptimisedMapping* list schedules the tasks, Figure 6.9(h), found through step A, Figure 6.8. However, with the chosen voltage scalings for the architecture processing cores, this mapping cannot satisfy the real-time constraint of 75ms. The *OptimisedMapping* swaps t_5 with t_6 in the fourth iteration as a neighbouring task mapping (step C, Figure 6.8) and gives the minimum number of SEUs experienced for the chosen voltage scaling, while meeting $T_{M_{ref}}$=75ms.

6.4.3 Iterative Assessment

With each set of voltage scaling coefficients resulting from the voltage scaling algorithm (step 1, Figure 6.5) soft error-aware application task mapping (step 2, Figure 6.5) is carried out to minimise the number of SEUs experienced through application task mapping. The resulting power consumption (P) and SEUs experienced (Γ) are then iteratively assessed using a score function, Z, to produce an optimised design in terms of minimised power consumption and improved reliability, such that real-time constraints are met (similar score function for joint optimisation is also used in [191]). The optimisation score function, Z, is defined by a linear combination of of normalised power consumption and number of SEUS experienced, given by

$$Z = 0.5\ Z_P + 0.5\ Z_\Gamma \quad , \tag{6.9}$$

where Z_P is the score related to power consumption (P), Z_Γ is the score related to reliability improvement in terms of total number of SEUs experienced (Γ) and 0.5 is the weighting factors for Z_P and Z_Γ (to give joint optimisation with equal weight to power consumption and reliability improvement). The Z_P value of a design is found by normalising power consumption (P, defined by (6.5)) due to the chosen voltage scaling (Section 6.4.1) and soft error-aware application task mapping (Section 6.4.2) by the maximum power consumption, P_{max} (given by (5.1) with the highest voltage settings,

i.e. s_i=1), Section 5.2.1, Chapter 5), i.e.

$$Z_P = \frac{P}{P_{max}} = \frac{C_L \sum_{i=1}^{\mathcal{C}} \alpha_i f_i(s_i) V_{dd_i}^2(s_i)}{C_L \sum_{i=1}^{\mathcal{C}} f_{max} V_{max_i}^2} \quad . \tag{6.10}$$

where f_{max} and V_{max} are maximum operating frequency and supply voltage (for example V_{max}=1V, f_{max}=200MHz for the given scaling options in Table 6.1). The score function related to SEUs experienced, Z_Γ, in (6.9) of the same design is defined by the ratio of SEUs experienced (Γ, given by (6.4)) to the maximum number of SEUs experienced (Γ_{max}, assuming maximum activity factor α=1 and highest voltage scaling, s_i=3, in (6.4)), i.e.

$$Z_\Gamma = \frac{\Gamma}{\Gamma_{max}} = \frac{T_M \sum_{i=1}^{\mathcal{C}} R_i \alpha_i \lambda_i}{T_M \sum_{i=1}^{\mathcal{C}} R_i \lambda_{max}} \quad . \tag{6.11}$$

where λ_{max} is the maximum soft error rate due to highest voltage scaling (for example V_{dd}=0.44V, f=66.7MHz for the given scaling options in Table 6.1). Using the optimisation score function, Z, defined in (6.9), the iterative assessment (step 3, Figure 6.5) identifies the design that gives the minimum Z value. Note that due to normalisation of the power consumption of a design, (P) by the maximum power consumption (P_{max}) in (6.10) Z_P is reduced for low power consumption. However, due to normalisation of SEUs experienced (Γ) by the maximum possible SEUs experienced (Γ_{max}) in (6.11) Z_Γ is increased for such design. Similarly, when power consumption increases Z_P is increased at the cost of reduced Z_Γ. As a result of this Z_P and Z_Γ relationship, the optimisation score function Z gives the minimum value for a design that gives the best trade-off between P and Γ. The design that gives minimum Z value and meets the real-time constraint is chosen as the optimised design. Figure 6.10 shows an example of iterative assessment using score function, Z, to effectively find an optimised design for the MPEG-2 video decoder (Figure 6.2) using MPSoC architecture with four processing cores (Figure 6.1). The horizontal axis shows the subsequent voltage scaling iterations arising from voltage scaling algorithm (Figure 6.6(a)) and vertical axis gives the Z_P, Z_Γ and Z values defined by (6.10), (6.11) and (6.9). As can be seen, the voltage scaling starts with scaling by 3 on each processing core (Figure 6.6(b)). For such low voltage scaling, Z_Γ is high with higher SEUs experienced (Γ) but Z_P but Z_P is low due to minimum power consumption. Higher Z_Γ results in a higher score function, Z, for this design. As the voltage scaling proceeds with next set of voltage scaling coefficients (Figure 6.6(b)), Z value also varies due to the trade-offs between Z_P and Z_Γ. Note that due to the voltage settings on processing cores (with V_{dd}=1V, f=200MHz on all

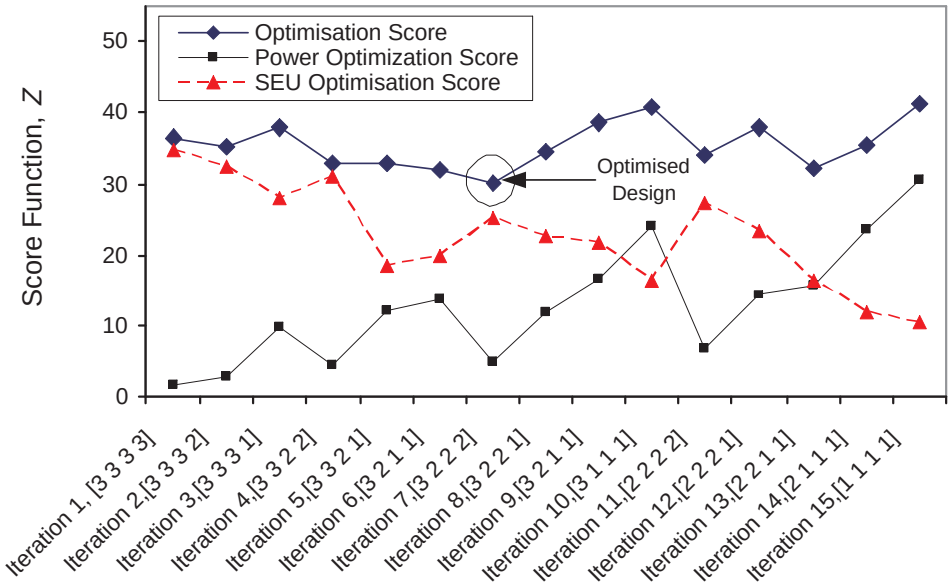


FIGURE 6.10: Example iterative assessment for design optimisation using MPEG-2 video decoder with four processing cores

processing core), design produced in iteration 15 results in the highest score function, Z. The design produced in iteration 7 (with V_{dd}=0.58V, f=100MHz on 3 processing cores and V_{dd}=0.44V, f=66.7MHz on a processing core) gives the best design in terms minimised Z value. Since real-time constraints are met for this design, it is returned as the optimised design for MPEG-2 video decoder (Figure 6.2) using MPSoC architecture with four processing cores (Figure 6.1).

6.5 Experimental Results

In this section, the effectiveness of the proposed soft error-aware design optimisation is evaluated using four experiments, Table 6.3. The experiments are carried out using MPEG decoder implemented with the architecture of Figure 6.1. The first three experiments, Exp:1, Exp:2 and Exp:3, are soft error-unaware optimisation with different design objectives using application task mapping obtained through simulated annealing [172]. Exp:4 is the proposed design optimisation. In all experiments, power minimisation is obtained through iterative voltage scaling (step 1, Figure 6.5) after application task mapping with an aim to meet the real-time constraint of decoding a *tennis* video sequence[1] of 437 frames at 29 frames per second (fps). The mapped tasks, voltage scaling on processing each core (s_i), per core execution time (T_i) and per core register usage (R_i) are shown in column 3-6, while the power consumption (P, mW), register usage (R, kbits/cyc), the multiprocessor execution time (T_M, clock cycles) and the number of

[1] ftp://ftp.tek.com/tv/test/streams/Element/

SEUs experienced (Γ) are given in columns 7-10 (Table 6.3). For each voltage scaling of the MPSoC processing cores, a time-limit of 30 minutes to search the design space is imposed. All experiments are carried out on an Intel(R) Core(TM)2 2GHz CPU running RHEL5. The number of SEUs experienced (column 7, Table 6.3) is found by fault injection technique (Section 6.2.2) assuming an arbitrary SER of 10^{-9} SEUs/bit/cycle (i.e. 1 SEU per 10ms for 1kb register bank). The power values are obtained by (6.1), with the α values found with the multiprocessor execution time (T_M) and execution times (T_i) of processing cores from Table 6.3 (note that switching activity factor, α=$\frac{T_i}{T_M}$).

Experiments	**MPSoC Core**	**Mapped Tasks**	**Voltage scaling**, s_i	**Execution time**, T_i, $\times 10^8$ **cyc.**	**Register usage,** R_i, *kb/cyc.*	**P**, mW	$R{=}\sum_i R_i$, *kb/cyc.*	T_M, $(\times 10^8)$ *cyc.*	Γ, $(\times 10^5)$
Exp:1 (Optimised for reduced register usage [172])	Core 1	t_1, t_2, t_3	2	14.4	17.3	9.53	80	38.1	3.46
	Core 2	t_4, t_5	2	21.4	16.4				
	Core 3	t_{11}	2	16.7	19.1				
	Core 4	t_6, t_7, t_8, t_9, t_{10}	1	28.8	27.2				
Exp:2 (Optimised for parallelism [172])	Core 1	t_1, t_2, t_3, t_4, t_9	3	24.6	37.7	4.04	114	28.6	5.22
	Core 2	t_8	3	18.5	19.4				
	Core 3	t_5, t_6, t_7	2	21.2	29.3				
	Core 4	t_{10}, t_{11}	2	20.1	28				
Exp:3 (Optimised for register usage &parallelism[172])	Core 1	t_8, t_9	3	23.1	23.6	4.15	92	30.9	4.18
	Core 2	t_{10}, t_{11}	2	20.7	27.9				
	Core 3	t_6, t_7	2	21.8	22				
	Core 4	t_1, t_2, t_3, t_4, t_5	2	17.9	19.1				
Exp:4 (Proposed optimisation)	Core 1	t_9	3	14.1	17	4.25	89	31.8	3.93
	Core 2	t_{10}, t_{11}	2	19.3	28				
	Core 3	t_7, t_8	2	25.8	20.4				
	Core 4	t_1, t_2, t_3, t_4, t_5, t_6	2	20.9	23.3				

TABLE 6.3: Comparison of soft error-unaware and the proposed soft error-aware optimisations using MPEG decoder MPSoC with four cores

Exp:1 demonstrates the impact of design optimisation with minimised register usage, R. As expected, the design produced gives the least register usage (R=$\sum_i R_i$) when compared to the other three experiments. The reduced R in Exp:1 is obtained at the expense of the highest multiprocessor execution time (T_M) of 38.1×10^8 clock cycles (as explained in Section 6.3). This makes it harder to scale down the voltages of the decoder cores. As a result, Exp:1 gives a design that has higher power consumption (9.53mW) than the optimised design produced in Exp:4 (4.25mW). However, the design produced in Exp:1 experiences lower SEUs than that in Exp:4 (3.46×10^5 SEUs compared 3.93×10^5 SEUs). This is because, the proposed design optimisation in Exp:4 gives lower voltages of the decoder cores, and hence lower power consumption compared to the design produced in Exp:1. The design produced in Exp:2 is optimised for high parallelism. This gives reduced multiprocessor execution time (T_M) of 28.6×10^8 clock cycles, which allows the voltages of the decoder processing cores to be scaled down. As a result, Exp:2 gives lower power consumption (4.04mW) than Exp:4 (4.25mW). Note that this reduction in multiprocessor execution time (T_M) in Exp:2 is achieved at the expense of the highest register usage (R=114kbits per cycle). Due to lower voltage scaling of the decoder cores and higher register usage, the design optimised for high parallelism, Exp:2, experiences the highest number of SEUs (Γ=5.22×10^5) when compared to the other three experiments. In Exp:3, the design has been optimised for both register usage (R) and high parallelism. Such optimisation gives a good trade-off between multiprocessor execution time and register usage, and minimises the product: $T_M \times R$ in (6.4). However, this does not necessarily minimise of the number of SEUs experienced since optimisation is carried using soft error-unaware task mapping. The design produced in Exp:4 employs soft error-aware task mapping (and minimises Γ in (6.4) by carefully mapping the tasks to minimise the product $T_i \times R_i$ on each core) and therefore gives less number of SEUs experienced than the design produced in Exp:3 (3.93×10^5 SEUs for Exp:4 compared to 4.18×10^5 SEUs for Exp:3). Note that, although the voltage scaling of the decoder cores are similar, the proposed design optimisation (Exp:4) gives about 3% higher power consumption compared to the design produced in Exp:3 due mapping of computation intensive tasks t_7, t_8 in core 3 and t_1-t_6 in core 4 of the decoder. For all design optimisation approximate number of SEUs experienced can also be found by (6.4) using the T_i and R_i values shown in columns 5 and 6 (Table 6.3).

To highlight the advantages of using the proposed design optimisation (Exp:4), Figure 6.11 shows comparison of power consumption (P) and SEUs experienced (Γ) of the decoder design in Exp:1, Exp:2 and Exp:3 compared to that of Exp:4. All experiments are carried out with same voltage scaling coefficients (s_1=2, s_2=2, s_3=3 and s_4=2) for an SER of 10^{-9}. As can be seen, the design produced in Exp:4 reduces the number of SEUs experienced by upto 38% compared to the optimised design in Exp:2, while consuming 9% lower power. When compared with the design produced in Exp:1, the optimised design in Exp:4 reduces SEUs experienced by 28%, while consuming only 7%

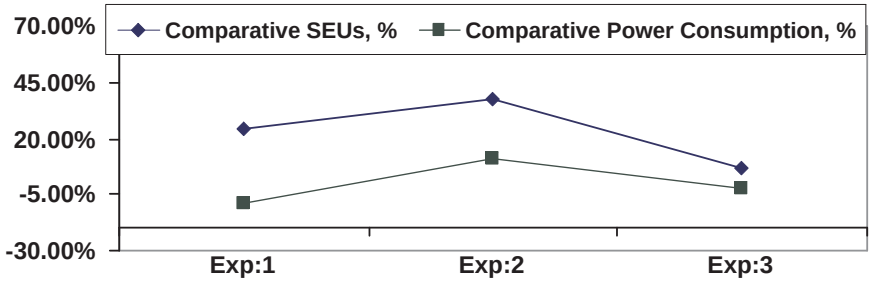


FIGURE 6.11: Comparison of power consumption (P, in mW) and SEUs experienced (Γ) of Exp:1, Exp:2 and Exp:3 when compared with Exp:4

higher power.

The design optimisations in Table 6.3 were carried out using MPEG-2 decoder. To demonstrate the effectiveness of the proposed design optimisation with other applications, random task graphs of 20, 40, 60, 80 and 100 tasks are also used. The random task graphs are generated using the random task and resource graph tool [179] (for sample task graphs, see Appendix C). The cost and the number of dependants in the random task graphs are generated using uniform probability distribution with computation cost between 1 and 30, communication cost between 1 to 10 (all costs as multiples of 3.5×10^6 clock cycles), task register usage between 1kbits to 5kbits and the number of dependants was found by exponential distribution between 0 to $N/2$, where N is the number of tasks. The deadline for random task graphs are set to 15, 20, 30, 40 and 50 seconds for random task graph with 20, 40, 60, 80 and 100 tasks, respectively. For these task graphs, the design optimisation is carried out with imposed time limits of 20, 30, 40, 50 and 60 minutes for 20, 40, 60, 80 and 100 tasks, respectively. Table 6.4 shows the results of using the proposed design optimisation (Exp:4) on the MPSoC using four processing cores (Figure 6.1) with the random task graphs. The voltage scalings on MPSoC processing cores, per core execution time (T_i) and per core register usage (R_i) are shown in columns 3-5. The power consumption (P), overall register usage (R), multiprocessor execution time (T_M) and the total number of SEUs experienced (Γ) are shown in columns 6-9 (Table 6.4).

Application	**MPSoC core**	**Voltage scaling,** s_i	**Exec. time,** T_i**,** $\times 10^8$ **cyc.**	**Reg. usage,** R_i*, kb/cyc.*	**P**, mW	$R{=}\sum_i R_i$, *kb/cyc.*	T_M,*cyc.* ($\times 10^8$)	Γ, ($\times 10^5$)
20 tasks	Core 1	3	8.3	19.1	4.34	66	12.8	2.27
	Core 2	3	8.4	14.7				
	Core 3	2	12.8	15.3				
	Core 4	2	12.4	17				
40 tasks	Core 1	3	10.9	26.9	5.2	90	23.6	2.87
	Core 2	3	9.8	20.1				
	Core 3	2	9.3	21.3				
	Core 4	1	29.8	21.2				
60 tasks	Core 1	2	13.6	25.5	5.1	107	31.2	4.82
	Core 2	2	12.4	27.9				
	Core 3	2	19.5	30.3				
	Core 4	2	30.3	23.6				
80 tasks	Core 1	3	15.5	25.2	4.4	129	41.3	6.13
	Core 2	3	15.2	38.7				
	Core 3	2	23.6	29.9				
	Core 4	1	46.4	35.8				
100 tasks	Core 1	3	15.9	32.4	4.8	149	53.7	8.25
	Core 2	3	28.7	33.6				
	Core 3	2	38.9	38.1				
	Core 4	2	49.6	44.3				

TABLE 6.4: Power consumption (P), register usage (R) and SEUs experienced (Γ) for different applications using Exp:4

As can be seen, depending on the application and its deadline the voltage scaling and the corresponding power consumption (P, in mW) vary (Table 6.4). However, the total register usage (R) arising from per core register usage (R_i) increases as the number of tasks in the random task graphs increases. Also, the multiprocessor execution time (T_M, in clock cycles) and per core execution time (T_i, in clock cycles) increase as the number of tasks in the task graphs increase. Due to the increased per core register usage (R) and per core execution time (T_i), the total number of SEUs experienced (Γ) also increases as the number of tasks in the random task graphs increases. For example, the random task graph with 100 tasks experiences the highest number of SEUs (i.e. 8.25×10^5 SEUs), while the random task graph with 20 tasks experiences the lowest number of SEUs (i.e. 2.27×10^5). Figure 6.12(a) and (b) show the comparisons of power consumption (P) and the number of SEUs experienced (Γ) using the design optimisations in Exp:3 and Exp:4 for the different random task graphs. As can be seen, the design optimisation in

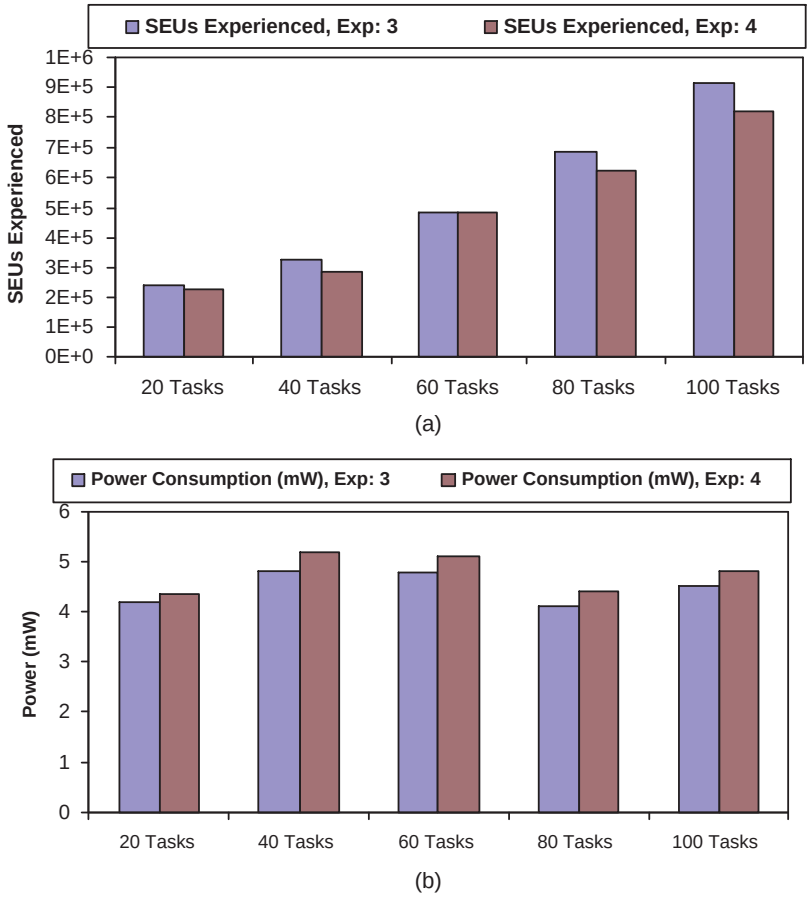


FIGURE 6.12: Comparison of power consumption (P, in mW) and SEUs experienced (Γ) between Exp:3 and Exp:4 for different random task graphs

Exp:4 consistently outperforms the design optimisation in Exp:3 in terms of the number of SEUs experienced (Γ) due to soft error-aware application task mapping carried out in Exp:4 (Section 6.4.2). For example, for the random task graph with 80 tasks the proposed design optimisation (Exp:4) reduces the SEUs experienced by 9.6% compared

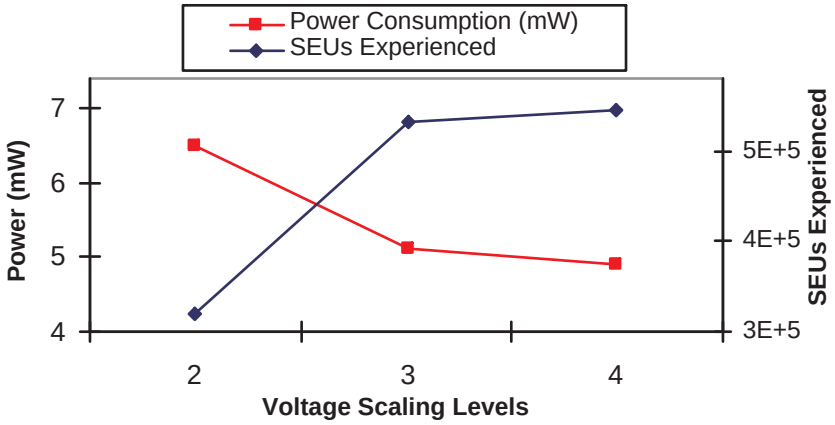


FIGURE 6.13: Power consumption (P, in mW) and SEUs experienced (Γ) for different scaling levels using the proposed design optimisation technique

to the design optimisation in Exp:3. This reduction in SEUs experienced is achieved with only 5% increase in the power consumption (P).

To show the impact of choice of voltage scaling levels, Figure 6.13 shows the power consumption (mW) and the number of SEUs experienced (Γ) by the optimised designs produced in Exp:4 with different voltage scaling levels. The design optimisations are carried out using MPSoC with four processing cores with random task graph of 60 tasks and employing the following voltage scaling levels: 2 levels (with 1V−200MHz, and 0.58V−100MHz), 3 levels (Table 6.1) and 4 levels (introducing 1.2V−236MHz in Table 6.1). As can be seen, with 4 scaling levels the proposed design optimisation (Exp:4) is able to minimise power further by 4% with only 3% increase in the number of SEUs experienced compared to 3 scaling levels. This is because with more scaling options, the power minimisation (step 1, Figure 6.5) has higher flexibility with more combinations of voltage scaling generated by the voltage scaling algorithm (Figure 6.6(a)). With 2 scaling levels, it is possible to reduce the number of SEUs experienced by 42% at the cost of 28% higher power consumption compared to 3 scaling levels due to limited voltage scaling options (Figure 6.13).

6.6 Architecture Allocation

Architecture allocation is a system-level design step for MPSoCs that deals with allocation of processing elements and their interconnections into the architecture (see Section 2.4.2.1, Chapter 2 for further details). In this work, architecture allocation is referred to as the allocation of number of processing cores in the MPSoC architecture. Table 6.5 shows the mapped tasks using the optimised mapping in Exp:4 for different allocations from two cores to six cores using MPEG-2 video decoder task graph

Allocation	**Core**	**Mapped Tasks**
2 Cores	Core 1	t_1, t_2, t_3, t_4, t_9, t_{10}, t_{11}
	Core 2	t_5, t_6, t_7, t_8
3 Cores	Core 1	t_1, t_2, t_3, t_4, t_5
	Core 2	t_6, t_7, t_8
	Core 3	t_9, t_{10}, t_{11}
4 Cores	Core 1	t_1, t_2, t_3, t_4, t_5, t_6
	Core 2	t_7, t_8
	Core 3	t_9
	Core 4	t_{10}, t_{11}
5 Cores	Core 1	t_1, t_2, t_3, t_4
	Core 2	t_5, t_6
	Core 3	t_7, t_8
	Core 4	t_9
	Core 5	t_{10}, t_{11}
6 Cores	Core 1	t_1, t_2, t_3, t_4
	Core 2	t_5, t_6
	Core 3	t_7
	Core 4	t_8
	Core 5	t_9
	Core 6	t_{10}, t_{11}

TABLE 6.5: Task distribution of MPEG-2 video decoder (Figure 6.2) among cores for different architecture allocations using the optimised task mapping in the proposed design optimisation technique (Figure 6.5)

(Figure 6.2). The architecture allocation is shown in column 1 and per core mapped tasks of the decoder task graph (Figure 5.7) are shown in columns 2-3 (Table 5.11). To demonstrate the impact of architecture allocation, Figure 6.14 shows the multiprocessor execution time (T_M, in clock cycles) and register usage (R, in kbits per cycle) using MPEG decoder MPSoCs. The voltage scaling of processing cores is carried out using three scaling levels (Table 6.1) and application task mapping is performed with the optimised mapping algorithm, *OptimisedMapping*, of the proposed design optimisation (Exp:4). The T_M and R are found while decoding a *tennis* video sequence of 437 frames at 29 frames per second. The architecture allocation is varied from two processing cores to six processing cores. As can be seen, with increase in the number of allocated cores, the register usage increases (Figure 6.14(a)). This is because with increased number of allocated cores the tasks mapping or distribution causes more duplication of the shared register resources among tasks. Also, as expected with increased number of allocated cores in the MPSoC architecture, the multiprocessor execution time (T_M) decreases with

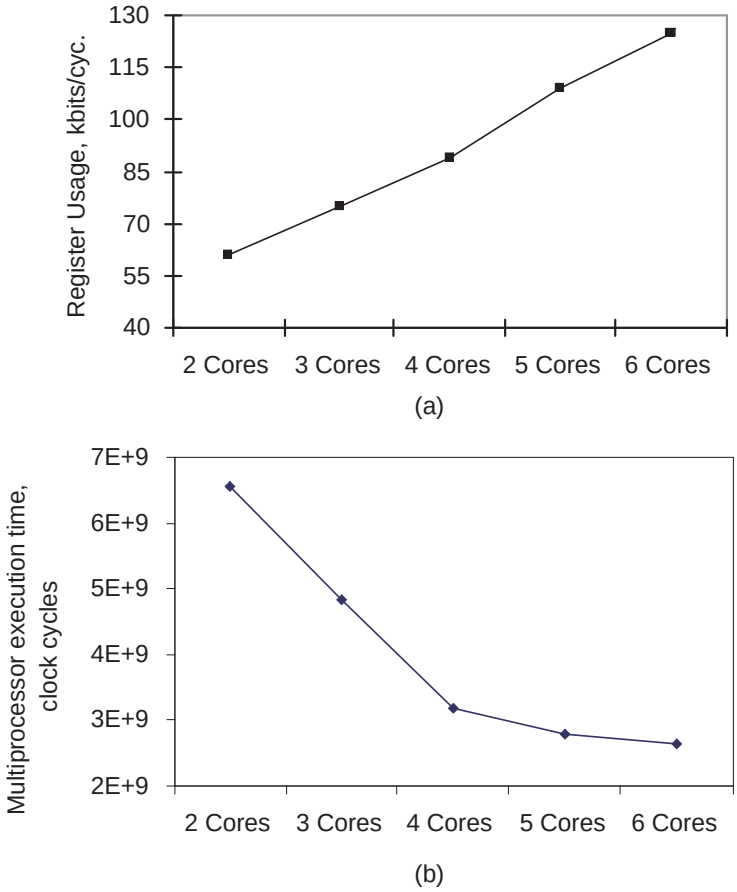


FIGURE 6.14: (a) Register usage (R, in kbits/cycle), and (b) multiprocessor execution time (T_M, in clock cycles) of the MPEG-2 decoder MPSoC for different architecture allocations

higher parallelism among the mapped tasks on processing cores (Figure 6.14(b)). Table 6.6 shows the impact of architecture allocation on the power consumption (P) and the number of SEUs experienced (Γ) using the optimised design produced in Exp:4. A number of applications, including MPEG decoder and random task graphs of 20, 40, 60, 80 and 100 tasks were used. The power consumption (P, in mW) and the number of SEUs experienced (Γ) for different architecture allocations are shown in columns 2-6 (Table 6.6). Two observations are made. Firstly, the architecture allocation with minimum power consumption (P) depends on the application and given real-time constraint. For example, in the case of the MPEG decoder, the least power consumption is found with four cores for the given real-time constraint of decoding *tennis* video sequence at 29fps. Secondly, with increased number of architecture cores, the number of SEUs experienced increases. The increased number of SEUs can be explained as follows. With higher number of cores, multiprocessor execution time (T_M) reduces and the overall register usage (R) increases (Figure 6.14). Due to reduced multiprocessor execution time, there is more opportunity for voltage scaling to reduce power consumption, which eventually increases the SER and the SEUs experienced. This is further exacerbated by the in-

creased register usage caused by distribution of tasks with increased number of cores in MPSoC architecture (Figure 6.14(a)). For example, the decoder with six processing cores experiences the highest number of SEUs, compared to the lowest for the decoder with 2 processing cores (row 2, Table 6.6). Similar observations for power consumption and the number of SEUs experienced are also observed with the random task graphs.

Application	**2 Cores**		**3 Cores**		**4 Cores**		**5 Cores**		**6 Cores**	
	P, mW	Γ $\times 10^5$	P, mW	Γ $\times 10^5$	P, mW	Γ $\times 10^5$	P, mW	Γ $\times 10^5$	P, mW	Γ $\times 10^5$
MPEG (11 tasks)	9.1	2.13	5.9	3.17	4.25	3.93	6.34	4.95	7.24	5.36
20 tasks	10.1	0.47	4.15	1.13	4.34	2.27	5.16	2.73	6.36	3.49
40 tasks	6.2	1.07	5.1	1.78	5.2	2.87	6.16	3.46	7.11	4.35
60 tasks	7.8	1.87	4.13	3.25	5.1	4.82	4.9	5.74	5.3	7.15
80 tasks	11.2	1.95	6.1	3.76	4.4	6.13	6.14	7.24	6.69	9.13
100 tasks	10.4	2.40	5.48	4.58	4.8	8.25	5.94	8.83	6.34	11.13

TABLE 6.6: Power consumption (P, in mW) and SEUs experienced (Γ, $\times 10^5$) for different applications and different architecture allocations

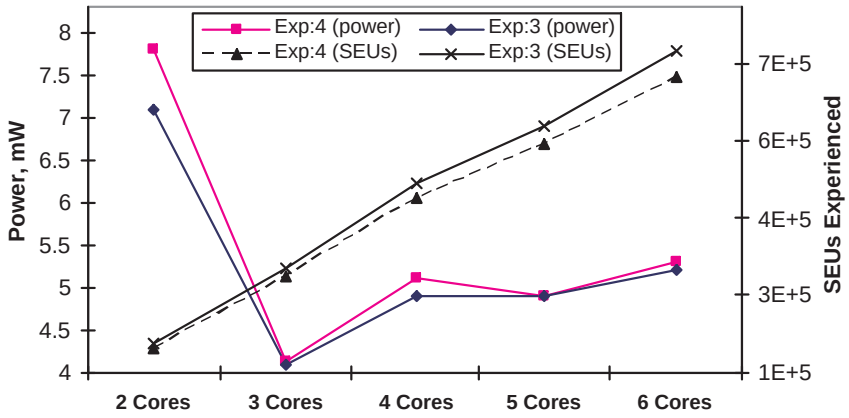


FIGURE 6.15: Comparison of power consumption (P, in mW) and SEUs experienced (Γ) between Exp:3 and Exp:4 for different architecture allocations using random task graph of 60 tasks

To compare between the soft error-aware and soft error-unaware design optimisations for different architecture allocations, Figure 6.15 shows the power consumption (P, in mW) and the SEUs experienced (Γ) by the optimised designs produced in Exp:4 and Exp:3 using the random task graph of 60 tasks. As can be seen, the proposed optimisation, Exp:4, consistently outperforms the design produced using joint optimisation of reduced R and high parallelism, Exp:3, with upto 7% reduction of SEUs experienced for an SER of 10^{-9}. This reliability improvement is achieved with only 3% higher power consumption using an MPSoC with six processing cores.

6.7 Concluding Remarks

This chapter has investigated the impact of application task mapping on the reliability of MPSoC (Section 6.3). Based on this investigation, a novel soft error-aware design optimisation has been proposed for low power and time-constrained MPSoCs (Section 6.4). The proposed design optimisation has been carried out using joint power minimisation using voltage scaling and reliability improvement in terms of minimised number of SEUs experienced through application task mapping. Using an MPEG decoder and random task graphs, it has been shown that the proposed optimisation technique can significantly reduce the number of SEUs experienced compared to soft error-unaware optimisation techniques, while power consumption is minimised and the real-time constraint is met (Section 6.5). Furthermore, the impact of architecture allocation on the power consumption and the number of SEUs experienced using the proposed optimisation technique has been examined (Section 6.6).

Chapter 7

Conclusions and Future Work

The overall aim of this research is to improve key aspects of multiprocessor system-on-chip (MPSoC) design with emphasis on performance, low power consumption and reliability. To meet this aim, an effective fault injection simulator is developed to enable reliability analysis and investigation into efficient and reliable on-chip communication architecture is carried out. Also, power minimisation techniques are developed for low power and reliable design. The reliability in the presence of soft errors is evaluated at application-level to reduce fault tolerance cost. A number of applications are used to validate the techniques developed in this work, including MPEG-2 video decoder and random task graphs. Section 7.1 presents a summary of research contributions made by this thesis and Section 7.2 outlines a number of worthy future research directions.

7.1 Summary and Research Contributions

Power minimisation is a prime objective in MPSoC design to extend battery life. Due to technology scaling and increased computational complexity of applications on these devices, scalable and efficient on-chip communication architectures are required for future MPSoCs. An emerging challenge in the design of MPSoCs is reliability in the presence of soft errors, particularly due to single-event upsets (SEUs) caused by radiation. Underpinning these MPSoC design issues and challenges, the following contributions have been made in this thesis:

SystemC Fault Injection Simulator : To facilitate reliability analysis and fault injection, a novel SystemC fault simulator was presented in Chapter 3. The fault simulator works by replacement of the original data and signal types to fault injection enabler types to initiate fault injection. Due to simple type replacement, the fault simulator merits from minimum design intrusion but yet high fault

representativeness when compared with some recently proposed fault simulation techniques. For validation and comparisons a number of example models were shown, including MPEG-2 video decoder. The fault injection simulator was used for evaluating the impact of SEUs in Chapters 4, 5 and 6.

Comparative Analysis of On-Chip Communication Architectures : In Chapter 4 comparative performance and reliability analysis was carried out between the on-chip communication architectures: advanced microprocessor bus architecture (AMBA) and emerging network-on-Chip (NoC); using MPEG-2 video decoder-based real application traffic. The performance comparison showed that NoC-based decoder outperforms AMBA-based decoder in terms of less operating frequency to decode a given video bitstream at specified frame rate due to higher core efficiency, concurrency and channel bandwidth. The reliability comparison showed that NoC-based decoder experiences less SEUs during computation due to less execution time but it experiences higher SEUs during communication due to higher register usage and channel latency. The impact of SEUs experienced in the decoders was also evaluated and compared at application-level using peak signal-to-noise ratio (PSNR) and frame error ratio (FER) metrics.

Voltage Scaling Technique for Power Minimisation : Chapter 5 established a relationship between power consumption and reliability in the presence of SEUs evaluating the impact of SEUs at application-level. Based on this relationship, a novel system-level power minimisation technique was proposed for MPSoCs to give optimised voltage scaling on processing cores. The aim was to generate designs that are optimised in terms of power consumption, while providing acceptable reliability at application-level and specified real-time decoding rate. To validate the proposed technique, MPEG-2 video decoder and synthetic examples were used. Using peak signal-to-noise ratio (PSNR) metric to evaluate the reliability of the decoder, it was shown that significant power reduction can be achieved using the proposed technique, while maintaining an acceptable PSNR and real-time performance. Furthermore, the impact of application task mapping and architecture allocation was investigated on the trade-offs between power consumption and reliability.

Soft Error-Aware Design Optimisation : Chapter 6 presented an investigation into the impact of application task mapping on reliability in terms of the number of SEUs experienced. Based on this study, a novel soft error-aware design optimisation technique was proposed using joint power minimisation through voltage scaling and reliability improvement through application task mapping. The aim was to minimise the number of SEUs experienced for a suitably identified voltage setting such that low power is consumed, while real-time constraints are met. The effectiveness of the proposed design optimisation was evaluated using a number of different applications, including MPEG-2 video decoder and random task graphs.

It was shown that the proposed soft error-aware design optimisation technique can effectively reduce the power consumption and improve reliability compared to soft error-unaware design optimisation techniques, while meeting a specified real-time constraint. Furthermore, the impact of architecture allocation on the proposed design optimisation technique was examined.

With the above contributions, this thesis focused on investigation into low power and reliable MPSoC design. The investigations carried out in the thesis were substantiated by a number of different experiments and comparisons involving different applications. It is hoped that the findings in this thesis would contribute towards current research efforts in efficient and appropriate design techniques for low power and reliable MPSoCs.

7.2 Future Research Directions

As part of future research, a number of worthy and interesting research challenges were identified. Two such research directions are shown below:

1. Soft error-aware leakage power minimisation, and
2. Online soft error-aware design optimisation.

The different aspects of these future research directions are described next.

7.2.1 Soft Error-Aware Leakage Power Minimisation

With technology scaling, leakage power of an integrated circuit is becoming increasingly important part of the total power consumption [177]. In Chapters 5 and 6, power minimisation has been carried out using dynamic power considerations. To achieve effective overall power minimisation, consideration of the leakage power is an interesting area of future research. An effective leakage power minimisation techniques is to incorporate adaptive body biasing (ABB) for DVS-enabled systems, which employs variable threshold voltage control through different body biasing conditions. However, reduction in leakage power through threshold voltage control directly affects soft error rate, as soft errors are induced by particle hits that can effectively collect enough charge to overcome threshold voltage to cause an upset [62]. As a result, reduction in leakage power causes increase in the number of soft errors. For example, it has been shown in [192] that using reverse body bias voltage of 0.5V worsens soft error rate by 36%. To achieve effective soft error-aware leakage power minimisation, the following aspects can be incorporated:

1. Formulation of power minimisation problem including ABB for DVS-enabled system and analysis of its effects on the reliability.

2. Establishing relationship between voltage scaling and application-level correctness considering the effect of ABB for DVS-enabled MPSoCs.

3. Development of a soft error-aware dynamic and leakage power minimisation technique employing DVS and ABB, while maintaining a specified application-level correctness and real-time performance.

7.2.2 Online Soft Error-Aware Design Optimisation

The voltage scaling techniques in Chapters 5 and 6 employ determination of supply voltages and operating frequencies with each processing core that are kept constant throughout the execution time. However, it is possible to achieve further power reduction using task-level voltage scaling assignment that allow runtime variations of supply voltages and operating frequencies. To employ such voltage scaling with an aim to achieve soft error-aware design optimisation, the impact of application task mapping on power minimisation and reliability need to be formulated. Based on such study, each task can be allocated a voltage setting to give optimised design in terms of minimised power consumption and minimised number of SEUs experienced. Also, with given real-time constraint, the voltage settings of application tasks can also be determined online using different techniques, such as workload prediction, worst-case execution time consideration. To evaluate the impact of online voltage scaling and task mapping on the reliability at application-level, the relationship between task-level voltage scaling and application-level correctness (Section 5.4.3, Chapter 5) needs to be revisited. A number of benefits of using such online voltage scaling has been demonstrated in detail in [193].

Appendix A

MPEG-2 Video Decoder

MPEG-2 video decoder constitutes a major component of MPSoC applications and has been used in Chapters 4, 5 and 6 as application case studies. Section A.1 presents the basics of MPEG-2 video decoder and Section A.2 shows example implementation of the decoder.

A.1 MPEG-2 Video Decoder Basics

MPEG-2 video decoder is an ISO standard for the generic coding of moving pictures [124]. It describes video decompression technique and is widely used in embedded multimedia systems, such as digital television, digital video discs (DVDs), digital video broadcasting (DVB), portable video devices, etc. In this work, MPEG-2 video decoder has been thoroughly used as an example of application-specific MPSoCs. Figure A.1 shows a block diagram of simplified video decoding process used in MPEG-2 standard.

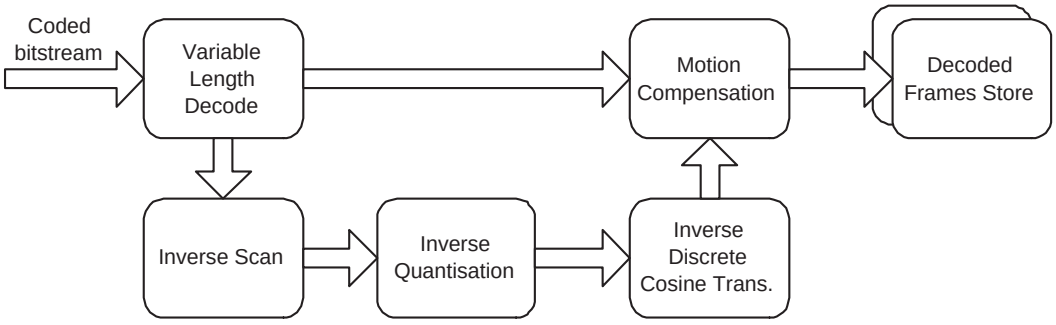


FIGURE A.1: Simplified MPEG-2 video decoding process

MPEG-2 video decoder takes encoded video bitstreams as inputs and generates decoded frames in specified format. As can be seen, the decoding process from coded bitstream is formed of five major processes (Figure A.1). The decoding is initiated by variable length

decoding process, followed by inverse scan, inverse quantisation, inverse discrete cosine transformation and motion compensation. After motion compensation, the decoding process is finished and the decoded video frames are stored in a frame buffer. In the following, the major processing steps and their inputs are briefly described.

- **Encoded Video Bitstream**: The coded video bitstream is the main input to the decoder, which is previously encoded by an MPEG-2 video encoder. Such encoder identifies the useful part of a raw video signal (called the entropy) and compresses it in a predefined format. The video encoding process is lossy process as some precision to the original video data are lost during compression due to time/frequency transformations and usage of fixed size value holders.

- **Variable Length Decoding**: The variable length decoding process splits the coded video bitstream in two sequences: header sequence and video sequence. The headers contain the necessary flags and parameters required to decode the video sequence and are arranged in predefined data structure and the video sequence contains the actual video data. The video is organised as groups of pictures (GOP) with different types of frames or pictures in sequence: intra (I), predicted (P) or bi-directional (B) picture, each picture with a hierarchical data structure. Figure A.2 shows hierarchical video data structure used in MPEG-2 video. As

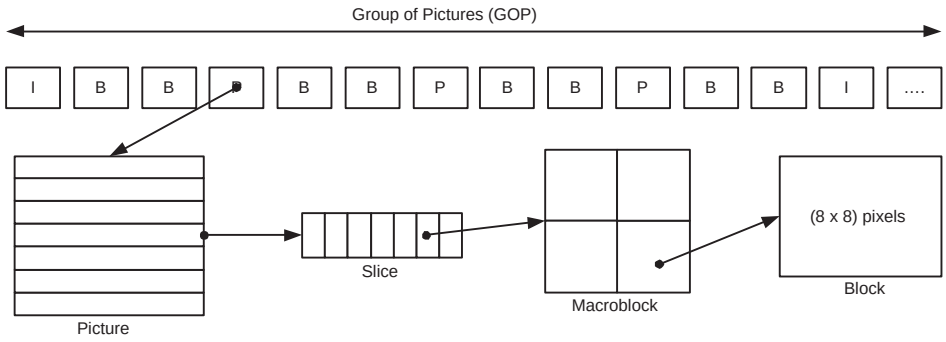


FIGURE A.2: Hierarchical video data structure used in MPEG-2 video

shown in Figure A.2, each picture contains slices, which are formed of contiguous macroblocks (MBs). Each macroblock is the 16 pixel segment, having four (8×8) blocks in each. Apart from decoding raw video into the hierarchical data structure, the variable length decoding process also performs necessary scaling and scanning as described in the different headers.

- **Inverse Scanning**: The quantisation matrices and video sequence blocks decoded by variable length decoder process are organised into one-dimensional array (i.e. $64 = 8 \times 8$). Inverse scan process transforms these arrays into two-dimensional arrays. The inverse scan patterns are defined in the header sequence.

- **Inverse Quantisation**: The resulting array of two-dimensional coefficients is inverse quantised to produce the reconstructed discrete cosine transformation (DCT) coefficients. This process is essentially a multiplication by the quantiser step size. The quantiser step size is modified by two mechanisms: weighting matrices and scale factor. As quantisation leaves some values to overflow, saturation is also carried out in the inverse quantisation process. The saturation process necessarily contains the values within the window, $[-2048 : 2047]$. During quantisation process, mismatch control is also performed by summing all of the reconstructed, saturated coefficients to determine and correct mismatches that may exist within the DCT coefficients.

- **Inverse Discrete Cosine Transformation**: Quantisation is followed by the inverse discrete cosine transformation (IDCT) process, which is similar to inverse Fourier transform, allowing the actual time domain picture information to be retrieved in a lossy manner. The DCT process is followed by sturation to contain within the range for coefficients as $[-2048 : +2047]$.

- **Motion Compensation and Storage / Display**: After IDCT process, the video blocks are picture ready but not in proper sequence in terms of time and space. Hence, the motion compensation process forms predictions from previously decoded pictures which are combined with the coefficient data (from the output of the IDCT process) to recover the final decoded samples. In the case where a block is not coded, either because the entire macroblock is skipped or the specific block is not coded, the decoded samples are derived through prediction. After the motion compensation is completed, the video blocks are then added and reconstructed to form uncompressed video (in sequence of frames). The decodes sequence of video frames are stored in the memory.

A.1.1 Frame Formats

Common intermediate format (CIF) is the standard video frame format commonly used in different video applications, such as video tele-conferencing, broadcast, etc. The sizes of other video frames are generally expressed as a function of the size of a CIF video frame. Table A.1 shows the different frame formats used in MPEG-2 video decoder with their resolution in number of pixels per frame. For example, sub-quarter CIF (SQCIF) has (128×96) pixels, while 2CIF has (352×576) pixels (Table A.1).

A.1.2 Decoding Rates

MPEG-2 video decoder has two different rates for video decoding: phase alternate line (PAL) and national television system committee (NTSC). Standard PAL decoding rate is

Format	Video Resolution (pixels)
SQCIF	128×96
QCIF	176×144
CIF	352×288
2CIF	352×576
4CIF	704×576
16CIF	1408×1152
DCIF	528×384

TABLE A.1: Different frame formats used in MPEG-2 video decoder

25 frames per second, while that of NTSC is 29.97 frames per second. The PAL standard has 20% more number of pixels in the vertical direction. For example, CIF format video in PAL decoding standard has (352×288) pixels, compared to (352×240) pixels in NTSC. MPEG video with different size and rates have been used in Chapters 4, 5 and 6.

A.1.3 MPEG Fidelity

The fidelity of decoded MPEG frames is often evaluated using various metrics. Peak signal-to-noise ratio (PSNR) is a popular metric used by [23, 24]. PSNR is found as the ratio of peak video signal power to the mean-squared-error or noise power as

$$PSNR = 10\log_{10}\frac{1}{ML}\sum_{m=1}^{M}\sum_{l=1}^{L}\frac{255^2}{(x_{m,l}-y_{m,l})^2} \quad . \tag{A.1}$$

where M is the number of frames coded in the video sequence, L is the number of pixels in a frame, $x_{m,l}$ and $y_{m,l}$ are the l-th pixels in m-th reference and decoded frames. Frame error ratio (FER) is another useful metric to describe how effectively MPEG-based systems can decode the frames and used in [145]. FER defines the ratio of erroneous or lost frames over total number of frames available as

$$FER = \frac{\text{Lost or Erroneous Frames}}{\text{Total Frames}}. \tag{A.2}$$

FER gives a measure of temporal quality in terms of effectives of decoding frames within a video bitstream or sequence. PSNR and FER has been used to evaluate reliability at application-level in Chapters 4 and 5.

A.2 MPEG-2 Video Decoder Implementation

In this work, two different MPEG-2 video decoder implementations have been used. In Chapter 4, MPEG-2 video decoder has been developed using behavioural modelling in a cycle accurate simulation environment in NIRGAM [25] (brief introduction to NIRGAM is presented in Appendix B, Section B.1) and in Chapter 5, C/C++ implementation of MPEG-2 video decoder has been ported to MPARM (brief introduction to MPARM is presented in Appendix B, Section B.2) with SystemC wrapper classes. Figure A.3 shows example SystemC prototype variable length decoder (VLD) core implementation used in Chapter 4. As shown in line 10, Figure A.3, each core extends the core interface in NIRGAM (refer to Appendix B, Section B.1), which requires implementation of the two processes, *send* and *recv* (line 50, 51, Figure A.3). The incoming data is fed by network interface (NI) to the core by *data_in* and outgoing data is fed to NI by *data_out*, each of which are structure of *packetdata* type (line 49, Figure A.3). The core activity is monitored by *MPEGMonitor* class (line 13) and mapping of local variables (line 15-26) is done through *MPEGMemory* class (line 12) through the constructor (line 47). The variable length decoder functionality is carried out through modular functions (line 27-44).

Figure A.4 shows example prototype MPEG-2 video decoder *initialiser* class used in MPARM environment (see Appendix B, Section B.2), which has the job of message queue definition and task allocation for a given architecture allocation of 4 cores. As can be seen, the prototypes for different tasks of Figure 5.7 are given in lines 6-10 and their definitions are given in lines 59-61 (Figure A.4). These taks are allocated to different processing cores using *Multiprocessing_configuration* RTEMS (real-time executive for embedded systems [163]) system variable in lines 35, 46 and 48 (Figure A.4. The different message queues for inter-processor communication are defined (lines 16-33, Figure A.4. These message queues are then mapped via interconnects through MPARM.

```
1  //=================VLD Header file=================
2  #ifndef _MPEGVLDecoder_H_
3  #define _MPEGVLDecoder_H_
4  #include "../../core/ipcore.h"
5  #include <fstream>
6  #include <string>
7  #include <math.h>
8  using namespace std;
9
10 class MPEGVLDecoder : public ipcore {
11         public:
12         MPEGMemory memory;
13         MPEGMonitor monitor;
14
15         sc_biguint<64> tempdata;
16         short int msb, intra_macroblock_count, non_intra_macroblock_count;
17         int isid, conid, block_count, scalable_mode, f_code[4][4], packet_no;
18         float total_wait_time, avg_wait_time, total_latency, avg_latency;
19         short int start_code, chroma_format;
20         int pict_scal, spatial_temporal_weight_code_table_index,
21                 macroblock_type, motion_vector_count, mv_format, dmv,
22                 h_r_size, v_r_size, vertical_size, picture_coding_type,
23                 picture_structure, pattern_code[12], block[12],
24                 dc_dct_pred[3], slice_no;
25         ...
26         ///Other VLD-related variables
27         ...
28         void read_a_packet(); void next_start_code();
29         void read_next_packet();
30         void write_cmd_packet(unsigned char, unsigned int);
31         void write_data_packet(int, unsigned int);
32         void write_packet(int, int, unsigned int);
33         void write_term_packet(unsigned int);
34         ....
35         ///Other MPEG-related function prototypes
36         ....
37         void sequence_header();
38         void extension_header();
39         void user_data_start_header();
40         void group_start_header();
41         void picture_start_header();
42         void slice_start_header();
43         long GetBits(unsigned char bits);
44         void Decode_MPEG2_Non_Intra_Block(int);
45         void Decode_MPEG2_Intra_Block(int);
46
47         /// Constructor
48         SC_CTOR(MPEGVLDecoder);
49
50         packetdata data_out, data_in;
51         void send();                    /// send flit/packet PROCESS
52         void recv();                    /// receive flit/packet PROCESS
53 };
54 #endif
55 //=================VLD Header End=================
56
```

FIGURE A.3: Example SystemC prototype class for MPEG variable length decoder (VLD) core used in Chapter 4

```
//=================Start of MPEG Init for RTEMS=================
#define TEST_INIT
#include "system.h"
#include <stdlib.h>

rtems_task decode_header_sequence(rtems_task_argument argument);
...
///Other tasks
...
rtems_task store_frame(rtems_task_argument argument);

rtems_task Init(rtems_task_argument ignored)
{
        rtems_status_code status;

        //Creating Message Queue for Task 1
        Queue_name[0] = rtems_build_name('Q', 'U', '1', ' ');
        status = rtems_message_queue_create(Queue_name[0], MAX_MES,
                sizeof(1024 * rtems_unsigned32), RTEMS_GLOBAL, &Queue_id[0]);
        directive_failed( status, "rtems_message_queue_create" );
        //Creating Message Queue for Task 2
        Queue_name[1] = rtems_build_name('Q', 'U', '2', ' ');
        status = rtems_message_queue_create(Queue_name[1], MAX_MES,
                sizeof(1024 * rtems_unsigned32), RTEMS_GLOBAL, &Queue_id[1]);
        directive_failed( status, "rtems_message_queue_create" );
        ....
        /// Creating more Message Queues for Task 3-10
        ....
        //Creating Message Queue for Task 11
        Queue_name[11] = rtems_build_name('Q', 'U', '1', '1');
        status = rtems_message_queue_create(Queue_name[11], MAX_MES,
                sizeof(1024 * rtems_unsigned32), RTEMS_GLOBAL, &Queue_id[11]);
        directive_failed( status, "rtems_message_queue_create" );

        if(Multiprocessing_configuration.node==1){
                Task_name[0] = rtems_build_name('0', '0', '1', ' ');
                status = rtems_task_create(Task_name[0], MAX_MES,
                        RTEMS_MINIMUM_STACK_SIZE, RTEMS_TIMESLICE,
                                RTEMS_LOCAL, &Task_id[0]);
        directive_failed( status, "rtems_task_create" );
        status = rtems_task_start(Task_id[0], (rtems_task_entry)
                decode_header_sequence, (rtems_task_argument)0);
        directive_failed(status, "rtems_task_start" );
  }
  ....
  ///Creating more Tasks and mapping tasks
  ....
        if(Multiprocessing_configuration.node==4){
                Task_name[11] = rtems_build_name('0', '0', '1', '1');
                status = rtems_task_create(Task_name[11], MAX_MES,
                        RTEMS_MINIMUM_STACK_SIZE, RTEMS_TIMESLICE,
                                RTEMS_LOCAL, &Task_id[11]);
        directive_failed( status, "rtems_task_create" );
        status = rtems_task_start(Task_id[11], (rtems_task_entry)
                store_frame, (rtems_task_argument)0);
        directive_failed(status, "rtems_task_start" );
  }
}
...
///Define all MPEG tasks and their communications
...
//=================End of MPEG Init for RTEMS=================
```

FIGURE A.4: Sample of MPARM *initialiser* class for defining MPEG-2 video decoder tasks highlighting task mapping to processing cores, used in Chapter 5

Appendix B

Simulation Tools Used

A number of simulation tools have been used throughout this research. Section B.1 presents NIRGAM simulation tool for NoC-based simulations [25], developed at University of Southampton and used in Chapter 4 and Section B.2 details MPARM simulation tool [21], which has been used for cycle-accurate power and performance profiling in Chapter 5. Simulated annealing tool ASA [178] is introduced in Section B.3 and Cocentric System Studio [67] (a cycle-accurate tool by Synopsys and used for AMBA-based simulations in Chapter 4) is described in Section B.4.

B.1 NIRGAM: NoC Interconnect Routing and Application Modelling

NIRGAM [25] is a SystemC-based discrete-event, cycle-accurate simulator for research on network-on-chip(NoC). It provides substantial support to experiment with NoC designs in terms of routing algorithms and applications on various topologies. Different topologies have been implemented in NIRGAM, which are i) 2-D mesh, and ii) 2-D torus. Different packet switching technique has been implemented, such as i) wormhole, and ii) store and forward. The following routing algorithms are also implemented i) deterministic XY, ii) adaptive odd-even(OE), and iii) source-based routing. The simulator provides different synthetic traffic generation techniques, suich as i) source (sender) traffic only, ii) sink/receiver-based traffic only, and iii) random traffic generator, etc. The traffic implementations can also be controlled using the following techniques: i) constant bit rate, ii) bursty, and iii) trace-based.

The simulator uses plug-in SystemC classes for applications and routers for easy customisation among different communication techniques as shown in Figure B.1. Different NoC parameters are made configurable from a global configuration file (Figure B.1). The different configurable paremeters are: i) topology size ($m \times n$), ii) clock frequency,

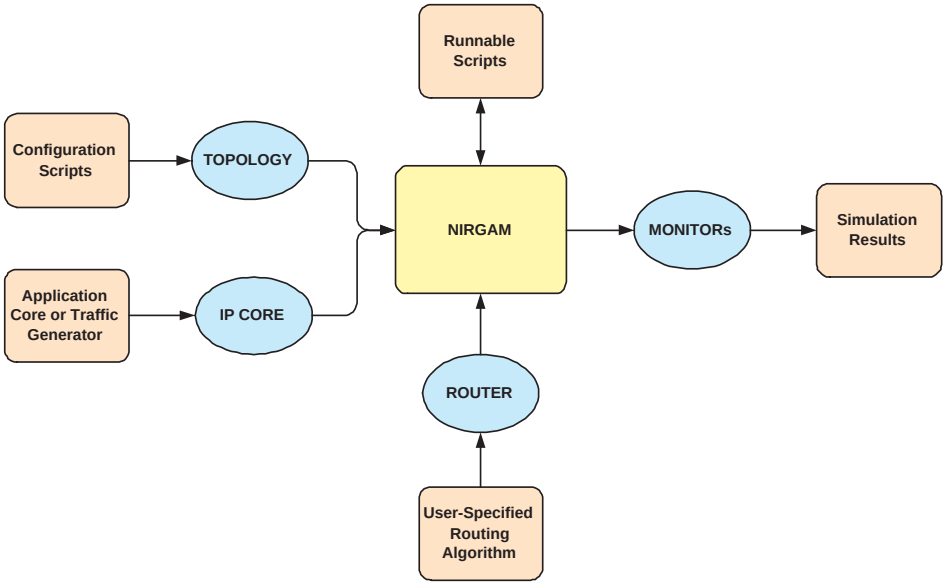


FIGURE B.1: Simplified block diagram of NIRGAM NoC simulator

iii) buffer depth, iv) packet type, v) flit size, and vi) virtual channel sizes, etc. The customisations can be carried out easily using global configuration file. The simulator is currently open source and is available at [25].

B.2 MPARM

MPARM [21] is a multi-processor cycle-accurate architectural simulator. Its facilitates system-level analysis of design trade-offs in the usage of different processors, interconnects, memory hierarchies and other devices. MPARM output includes accurate profiling of system performance, execution traces, signal waveforms, and, for many modules, power estimation. The OCP 2.0 point-to-point link is deployed to connect system components and xPIPES [87] is incorporated for NoC-based simulations. The MPARM platform includes hardware components and software components. Currently, a port of the RTEMS [163] operating system runs on MPARM as shown in Figure B.2 . The MPARM used in this work uses software instruction set simulation (ISS) platform for ARM (called SWARM), implementing functionalities of ARM7TDMI [194] processor. The RTEMS environment gives easy integration between software implementation using C/C++ and hardware implementation using SystemC wrapper classes for interconnects and other hardware components (Figure B.2). This makes MPARM a good platform for easy HW/SW codesign. The architecture allocation is specified within MPARM using configuration scripts and application task mappings can be easily carried out using RTEMS task mapping with semaphore-based resource sharing between tasks. Chapter 5 uses MPARM to employ voltage scaling in MPSoC cores and obtain power consumption results.

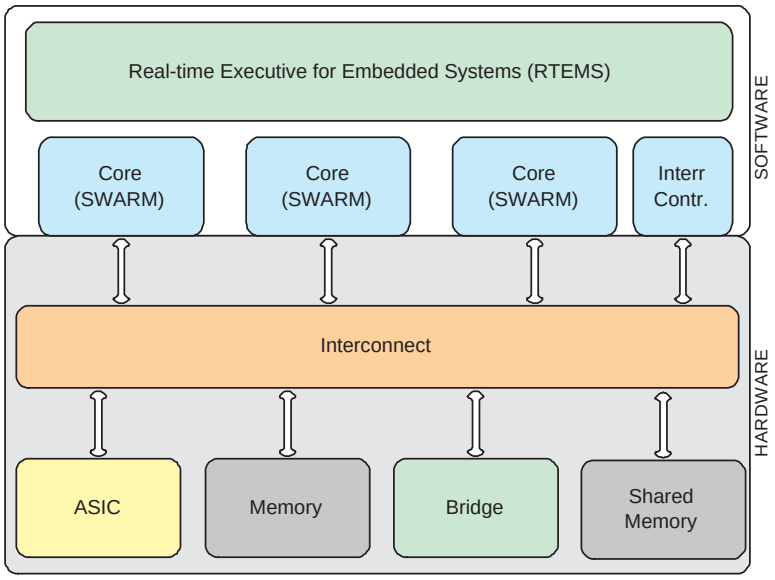


FIGURE B.2: Simplified block diagram of MPARM hardware/software codesign

B.3 Adaptive Simulated Annealing Tool

Adaptive Simulated Annealing (ASA) [178] is a C-language code developed to statistically find the best global fit of a non-linear constrained non-convex cost-function over a D-dimensional space. This algorithm permits an annealing schedule for "temperature" T decreasing exponentially in annealing-time k, $T = T_0 exp(-ck^1/D)$. The introduction of re-annealing also permits adaptation to changing sensitivities in the multi-dimensional parameter-space. This annealing schedule is faster than fast Cauchy annealing, where $T = T_0/k$, and much faster than Boltzmann annealing, where $T = T_0/\log_e k$. ASA has over 100 OPTIONS to provide robust tuning over many classes of non-linear stochastic systems.

ASA is widely used tool for simulated annealing experiments due to its easy customisation and access. The tool requires definition of the *cost* function that would be optimized. The mapping optimisation is carried out in this work (Chapters 5 and 6) using group migration-based task movement algorithm [172], which is used to map task graphs on multiple processing cores in a greedy fashion. Group migration is easy to implement and suitable for automatic architecture exploration and application task mapping [172]. For application task mapping, the *cost* function is considered as a equally weighted parameters with low memory consumption and/or high parallelism (i.e. low multiprocessor execution time). The *cost* function is found by these two factors from a mapped system by scheduling a given task graph. The optimisation starts with all application tasks mapped on a single core among a given number of allocated cores. In each iteration, a new movement is carried out to improve the cost function until improvement is minimal in subsequent moves. The improvement of cost function is characterised by temperature

cooling function within the ASA tool. In this work, the initial temperature is assumed to be 1.0 and the final temperature is expected to be 0.0001 using temperature proportion of 0.95.

B.4 Cocentric System Studio

The CoCentric System Studio [67] by *Synopsys* is an integrated environment for SystemC-based hardware description and simulations. The tool provides graphical interface of SystemC hardware or software classes and monitors, making it easier to carry out simulation-specific tasks. It employs object oriented modeling throughout using pre-built SystemC classes. The System Studio offers a wide variety of modelling capabilities, providing designer with the means to capture complex systems quickly and efficiently. The modelling paradigms supported can be hierarchically mixed at all levels, making System Studio an extremely versatile system modelling platform. The tool also contains a number of pre-built models to facilitate simulations, such as AMBA AHB, AHB master, AHB slave, arbiter and decoder. In Chapter 4 Cocentric System Studio has been used to develop AMBA AHB shared-bus architecture for comparison with network-on-chip (NoC).

Appendix C

Random Task and Resource Graphs

In this research, a number of random task graphs have been used as synthetic application examples to help evaluate the effectiveness of the design techniques proposed in Chapters 5 and 6. These task graphs are generated using the random task and resource graph tool [179], specifically developed for this research. In this chapter, a number of different random task graphs are shown. To demonstrate how the task graphs are described, example task graph with 10 tasks are shown with and without the associated register resources in Sections C.1 and C.2. Later, example descriptions of task graph with 20 tasks, 40 tasks are shown in Section C.3.

C.1 Example Task Graph: *10 Tasks without Resource*

The following task graph description is a random task graph with 10 tasks without showing resource mapping, i.e. the register usage of each task. The description is organised as follows:

1. The first line contains the number of tasks in the task graph.
2. Each following line contains the task description. Each task is denoted by it's number followed by the computation cost.
3. The number followed by the computation cost is the number of dependants of the task. This is then followed by the dependant nodes as pairs with the dependant node number and communication cost.
4. The last node has 0 number of dependencies as expected.

The cost and the number of data dependencies in the random task graphs were generated using uniform probability distribution with computation cost between 1 and 40, communication cost between 1 to 10 (all costs as multiples of 3.5×10^6 clock cycles). The number of dependencies were given by exponential distribution between 0 to $N/2$, where N is the number of tasks. Figure C.2 shows the diagrammatic representation of

```
1  //=================Start of Task Graph with 10 Tasks=================
2  10
3  1 34 3 2 2 3 5 4 4
4  2 16 1 3 4
5  3 38 3 4 9 5 6 6 5
6  4 15 2 5 2 6 1
7  5 31 3 6 7 7 1 8 7
8  6 13 3 7 2 8 3 9 1
9  7 26 1 8 3
10 8 16 2 9 2 10 4
11 9 15 1 10 4
12 10 19 0
13 //=================End of Task Graph with 10 Tasks=================
14
```

FIGURE C.1: Example task graph description with 10 tasks without resource mapping

the task graph with 10 tasks shown in Figure C.1. The nodes in Figure C.2 describe

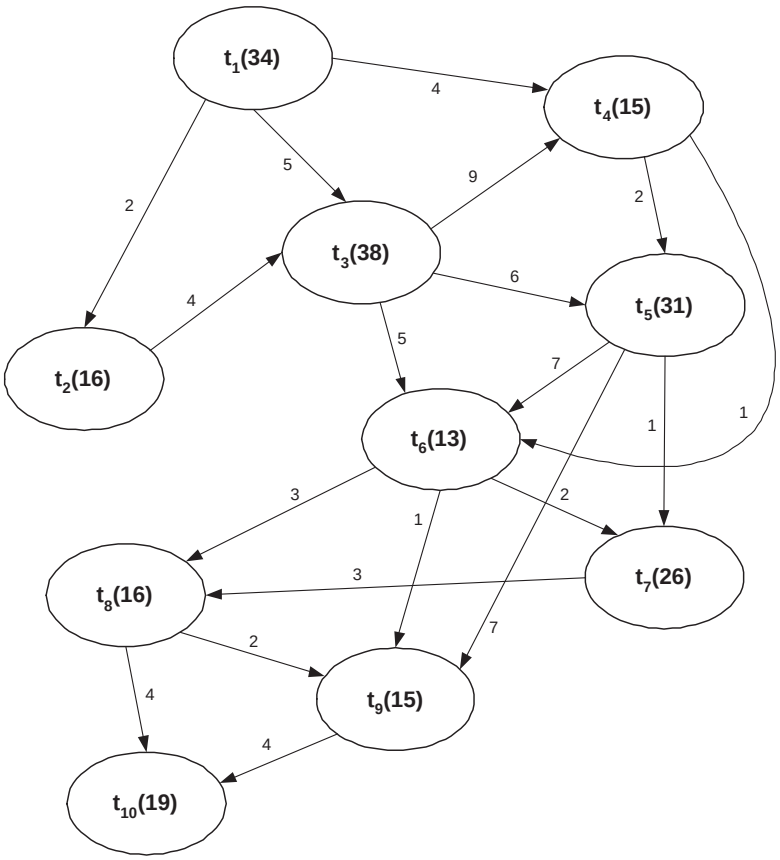


FIGURE C.2: Example task graph with 10 tasks without resource mapping showing computational tasks as nodes and communication tasks as edges

the name of the task (for example, t_1 meaning task 1). The numbers within the nodes describe the computational cost (for example, t_2 has computational cost of 16). The arrows between nodes show the direction of communication and the numbers on them describe the communication cost between tasks (the communication cost between t_1 and t_2 is 2). Next, a random task graph with 10 tasks is shown with mapped register resources.

C.2 Example Task Graph: *10 Tasks with Resource*

The following task graph description is a random task graph with 10 tasks without showing resource mapping, i.e. the register usage of each task. The description is organised as follows:

1. The first line contains the number of register usage components.
2. Each line is then followed by the register usage.
3. The line following contains the number of tasks in the task graph.
4. Each following line contains the task description. Each task is denoted by it's number followed by the computation cost.
5. The number followed by the computation cost is the number of dependencies of the task. This is then followed by the node numbers of dependants and communication costs as pairs.
6. It is then followed by the number of register resource components, followed by their numbers.
7. The last node has 0 number of dependencies as expected.

The cost and the number of data dependencies in the random task graphs were generated using uniform probability distribution with computation cost between 1 and 40, communication cost between 1 to 10 (all costs as multiples of 3.5×10^6 clock cycles). The task register usage is generated using uniform distribution between 1kbits to 20kbits and the number of dependencies were given by exponential distribution between 0 to $N/2$, where N is the number of tasks. Figure C.4 shows the diagrammatic representation of the task graph with 10 tasks shown in Figure C.3. As shown in Figure C.2, a set of register resources are associated with each node. For example, task t_1 has register resources r_1 and r_2, while task t_3 has register resources r_1, r_2, r_3 and r_4. These two tasks share the registers r_1 and r_2 between them.

```
1  //=================Start of Task Graph with 10 Tasks=================
2  7
3  20480
4  4096
5  6144
6  19456
7  5120
8  10240
9  20480
10 10
11 1 34 3 2 2 3 5 4 4 2 1 2
12 2 16 1 3 4 2 2 1
13 3 38 3 4 9 5 6 6 5 4 3 4 1 2
14 4 15 2 5 2 6 1 4 4 1 2 3
15 5 31 3 6 7 7 1 8 7 2 1 2
16 6 13 3 7 2 8 3 9 1 2 2 1 4
17 7 26 1 8 3 5 2 3 4 5 1
18 8 16 2 9 2 10 4 4 4 1 2 3
19 9 15 1 10 4 2 1 2 6
20 10 19 0 4 2 4 6 7
21 //=================End of Task Graph with 10 Tasks=================
```

FIGURE C.3: Example task graph description with 10 tasks with resource mapping

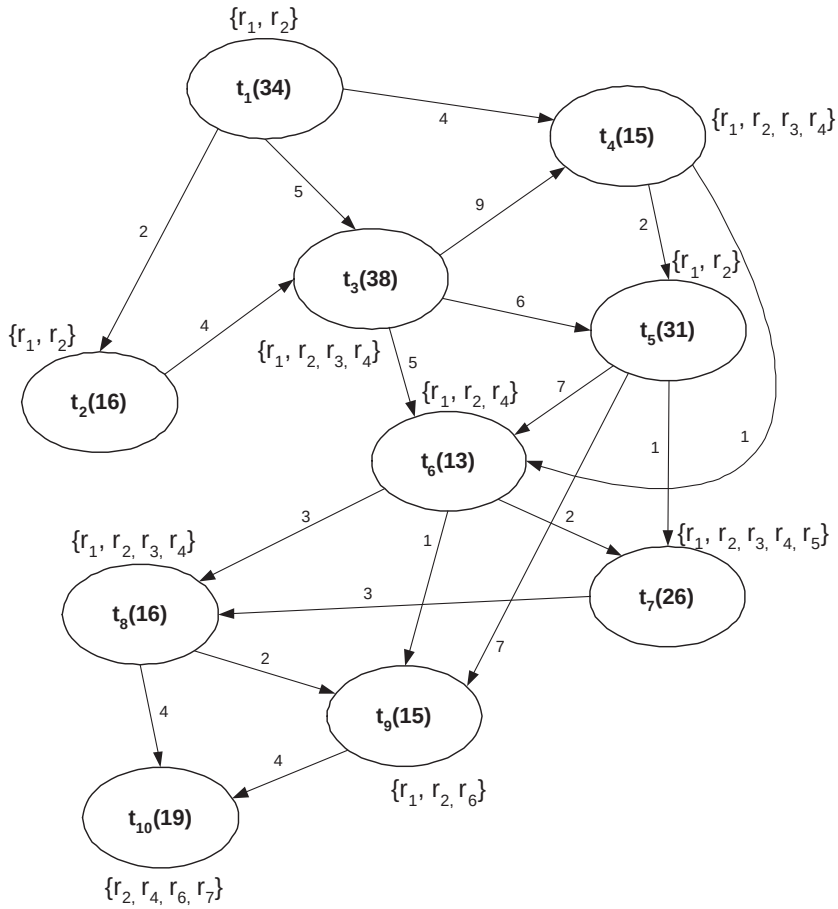


FIGURE C.4: Example task graph with 10 tasks with resource mapping showing computational tasks as nodes, communication tasks as edges and register resources with each task

C.3 Other Example Task Graphs

Figure C.5 shows an example task graph with 20 tasks with associated register resources. A total of 10 registers are used.

```
1  //=================Start of Task Graph with 20 Tasks=================
2  10
3  20480
4  4096
5  6144
6  19456
7  5120
8  10240
9  20480
10 15360
11 11264
12 15360
13 20
14 1 38 3 2 13 3 10 4 7 7 1 2 3 4 5 6 7
15 2 15 2 3 2 4 1 7 2 3 4 5 6 7 1
16 3 31 3 4 10 5 2 6 10 2 1 2
17 4 13 3 5 3 6 5 7 2 3 1 2 3
18 5 26 1 6 5 10 5 6 7 8 9 10 1 2 3 4
19 6 16 2 7 2 8 6 8 6 7 8 1 2 3 4 5
20 7 15 2 8 1 9 5 6 1 2 3 4 5 6
21 8 15 3 9 11 10 6 11 3 6 2 3 4 5 6 1
22 9 28 3 10 10 11 7 12 2 5 4 5 1 2 3
23 10 12 2 11 7 12 6 8 2 3 4 5 6 7 8 1
24 11 28 4 12 6 13 12 14 2 15 11 9 2 3 4 5 6 7 8 9 1
25 12 21 3 13 5 14 1 15 9 7 5 6 7 1 2 3 4
26 13 24 3 14 3 15 1 16 5 3 1 2 3
27 14 17 2 15 7 16 9 7 7 1 2 3 4 5 6
28 15 22 3 16 11 17 9 18 10 2 1 2
29 16 13 1 17 12 3 1 2 3
30 17 19 1 18 2 7 3 4 5 6 7 1 2
31 18 30 1 19 7 10 8 9 10 1 2 3 4 5 6 7
32 19 36 1 20 2 6 1 2 3 4 5 6 10
33 20 21 0 10 1 2 3 4 5 6 7 9 10
34 //=================End of Task Graph with 20 Tasks=================
```

FIGURE C.5: Example task graph description with 20 tasks with resource mapping

Figure C.6 shows an example task graph with 40 tasks with associated register resources. A total of 19 registers are used.

```
1  //=================Start of Task Graph with 40 Tasks=================
2  19
3  20480
4  4096
5  6144
6  19456
7  5120
8  10240
9  20480
10 15360
11 11264
12 15360
13 16384
14 14336
15 7168
16 17408
17 11264
18 4096
19 10240
20 8192
21 5120
22 40
23 1 18 1 2 2 12 1 2 3 4 5 6 7 8 9 10 11 12
24 2 12 2 3 9 4 3 10 2 3 4 5 6 7 8 9 10 1
25 3 14 2 4 7 5 2 10 3 4 5 6 7 8 9 10 1 2
26 4 11 2 5 5 6 2 11 4 5 6 7 8 9 10 11 1 2 3
27 5 32 2 6 2 7 5 13 5 6 7 8 9 10 11 12 13 1 2 3 4
28 6 30 3 7 5 8 1 9 4 3 3 1 2
29 7 23 2 8 4 9 7 13 7 8 9 10 11 12 13 1 2 3 4 5 6
30 8 39 2 9 8 10 2 16 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7
31 9 34 2 10 6 11 4 2 1 2
32 10 28 3 11 5 12 7 13 2 2 2 1
33 11 19 1 12 3 9 2 3 4 5 6 7 8 9 1
34 12 23 3 13 6 14 4 15 6 15 12 13 14 15 1 2 3 4 5 6 7 8 9 10 11
35 13 29 3 14 1 15 2 16 1 17 13 14 15 16 17 1 2 3 4 5 6 7 8 9 10 11 12
36 14 13 2 15 2 16 2 14 14 1 2 3 4 5 6 7 8 9 10 11 12 13
37 15 30 1 16 4 20 15 16 17 18 19 20 1 2 3 4 5 6 7 8 9 10 11 12 13 14
38 16 36 1 17 5 9 7 8 9 1 2 3 4 5 6
39 17 37 1 18 7 13 4 5 6 7 8 9 10 11 12 13 1 2 3
40 18 39 2 19 8 20 9 12 6 7 8 9 10 11 12 1 2 3 4 5
41 19 17 3 20 6 21 3 22 2 10 9 10 1 2 3 4 5 6 7 8
42 20 14 4 21 2 22 6 23 1 24 4 17 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 1 2
43 21 18 3 22 3 23 4 24 7 18 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 1 2
44 22 29 1 23 2 11 11 1 2 3 4 5 6 7 8 9 10
45 23 33 2 24 9 25 9 8 7 8 1 2 3 4 5 6
46 24 16 2 25 6 26 7 13 11 12 13 1 2 3 4 5 6 7 8 9 10
47 25 12 4 26 7 27 3 28 3 29 9 10 5 6 7 8 9 10 1 2 3 4
48 26 20 1 27 8 2 2 1
49 27 14 1 28 6 15 12 13 14 15 1 2 3 4 5 6 7 8 9 10 11
50 28 28 4 29 5 30 5 31 5 32 7 13 2 3 4 5 6 7 8 9 10 11 12 13 1
51 29 30 4 30 1 31 3 32 8 33 3 8 5 6 7 8 1 2 3 4
52 30 19 4 31 8 32 7 33 5 34 4 6 6 1 2 3 4 5
53 31 12 2 32 1 33 3 9 4 5 6 7 8 9 1 2 3
54 32 23 2 33 4 34 2 17 15 16 17 1 2 3 4 5 6 7 8 9 10 11 12 13 14
55 33 14 2 34 2 35 9 14 5 6 7 8 9 10 11 12 13 14 1 2 3 4
56 34 14 2 35 4 36 4 15 4 5 6 7 8 9 10 11 12 13 14 15 1 2 3
57 35 18 3 36 5 37 2 38 2 14 7 8 9 10 11 12 13 14 1 2 3 4 5 6
58 36 31 4 37 7 38 6 39 7 40 9 13 10 11 12 13 1 2 3 4 5 6 7 8 9
59 37 33 4 38 6 39 7 40 6 41 9 5 2 3 4 5 1
60 38 20 0 4 2 3 4 1
61 39 28 1 40 9 20 19 20 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
62 40 12 0 11 7 8 9 10 11 1 2 3 4 17 19
63 //=================End of Task Graph with 40 Tasks=================
```

FIGURE C.6: Example task graph description with 40 tasks with resource mapping

Bibliography

[1] L. Ross, editor. *Future of Embedded Systems Technology.* BCC Research, July 2005.

[2] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965. Former Director, Fairchild Semiconductors.

[3] G. E. Moore. Progress in digital integrated electronics. *IEDM Technical Digest*, pages pp.11–13, 1975.

[4] W. Wolf. The future of multiprocessor systems-on-chips. In *Design and Automation Conference (DAC)*, pages 681–685, 2004.

[5] G. Martin. Overview of the MPSoC design challenge. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, pages 274–279, 2006.

[6] H.G. Lee, N. Chang, U.Y. Ogras, and R. Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):1–20, August 2007.

[7] M. A. Breuer. Multimedia applications and imprecise computation. In *Proceedings of 8th Euromicro Conference on Digital System Design*, pages 2–7, 2005.

[8] M. A. Breuer and H. H. Zhu. Error tolerance and multimedia. In *Proceedings of International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 521–524, Dec 2006.

[9] F. Dabiri, A. Nahapetian, M. Potkonjak, and M. Sarrafzadeh. *Lecture Notes in Computer Science: Power and Timing Modeling, Optimization and Simulation*, volume 4644, chapter Soft Error-Aware Power Optimization Using Gate Sizing, pages 255–267. Springer, 2007.

[10] S. Mitra, P. Sanda, and N. Seifert. Soft errors: Technology trends, system effects, and protection techniques. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[11] M. Olmos. Radiation results of the SER test of Actel, Xilinx and Altera FPGA instances. Technical report, iRoC, October 2004.

[12] J. F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, 1996. IBM Corp., USA.

[13] G. Chen, M. Kandemir, and E. Li. Energy-aware computation duplication for improving reliability in embedded chip microprocessors. In *Asian and South Pacific Design Automation Conference*, pages 134–139, Yokohama, Japan, 2006.

[14] F. Dabiri, N. Amini, M. Rofouei, and M. Sarrafzadeh. Reliability-aware optimization for DVS-enabled real-time embedded systems. In *ISQED '08: Proceedings of the 9th international symposium on Quality Electronic Design*, pages 780–783, Washington, DC, USA, 2008. IEEE Computer Society.

[15] R. R. Tamhankar, S. Murali, and G. De-Micheli. Performance driven reliable link design for networks on chips. In *Proceedings of the Conference on Asia South Pacific Design Automation*, pages 749–754, 2005.

[16] Y. Xie, L. Li, M. Kandermir, N.Vijaykrishnan, and M.J. Irwin. Reliability-aware co-synthesis for embedded systems. *The Journal of VLSI Signal Processing*, 49(1):87–99, 2004.

[17] L. Benini and D. Bertozzi. Network-on-chip architectures and design methods. In *IEE Proceedings of Computers and Digital Techniques*, March 2005.

[18] D. Flynn. AMBA: Enabling reusable on-chip design. *IEEE Micro*, 17(4):20–27, Jul/Aug 1997.

[19] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38, March 2006.

[20] Arteris. A Comparison of Network on Chip and Buses. http://www.arteris.com/noc_whitepaper.pdf.

[21] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the multi-processor SoC design space with SystemC. *The Journal of VLSI Signal Processing*, 41(2):169–182, September 2005.

[22] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Cost considerations in network on chip. *Special issue on Networks on chip and Reconfigurable Fabrics of the VLSI Journal of Integration, Elsevier Science Publishers B. V.*, 38(1):19–42, October 2004.

[23] X. Li and D. Yeung. Application level correctness and its impact on fault tolerance. In *Proceedings of 13th International Symposium of High Performance Computer Architecture*, pages 181–192, Phoenix, AZ, USA, February 2007.

[24] X. Li and D. Yeung. Exploiting application-level correctness for low-cost fault tolerance. *Journal of Instruction-level Parallelism*, 10:1–28, 2008.

[25] L. Jain. NIRGAM: A dynamic SystemC simulator for NoC. http://nirgam.ecs.soton.ac.uk, (last accessed on 23/04/2010).

[26] Synposys Inc. PrimeTime. http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx, (last accessed on 23/04/2010).

[27] AMBA. Advanced microprocessor bus architecture specification, v2.0, 1999. http://www.arm.com.

[28] K. K. Ryu, E. Shin, and V.J. Mooney. A comparison of five different multiprocessor SoC bus architectures. In *Proceedings of Euromicro Symposium on Digital Systems, Design*, volume 1, pages 202–209, 2001.

[29] S. Lee and C. Lee. A high performance SoC on-chip-bus with multiple channels and routing processes. In *Proc. IFIP International Conference on Very Large Scale Integration*, pages 86–91, 16–18 Oct. 2006.

[30] P. Yew and J. Xue, editors. *Advances in Computer Systems Architecture: 9th Asia-Pacific Conference, ACSAS*. Springer, 2004. ch: A Switch Wrapper Design for SNA On-Chip-Network.

[31] G. Ma and H. He. Design and implementation of an advanced DMA controller on AMBA-based SoC. In *Proc. IEEE 8th International Conference on ASIC ASICON '09*, pages 419–422, 20–23 Oct. 2009.

[32] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computing*, 39(6):775–785, June 1990.

[33] H.G. Lee, U. Y. Ogras, R. Marculescu, and N. Chang. Design space exploration and prototyping for on-chip multimedia applications. In *Proceedings of the Design Automation Conference (DAC)*. San Francisco, California, USA, July 24-28 2006.

[34] U. Ogras, J. Hu, and R. Marculescu. Key research problems in NoC design: A holistic perspective. In *Proceedings of CODES+ISSS*, pages 69–74, 2005. Jersey City, NJ, USA.

[35] N. Banerjee, P. Vellanki, and K.S. Chatha. A power and performance model for network-on-chip architectures. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE), IEEE*, pages 1250–1255, 2004.

[36] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey. A 1 V heterogeneous reconfigurable processor IC for baseband wireless

applications. In *Digest of Technical Papers of International Solid-State Circuits Conference (ISSCC), IEEE*, pages 68–69, 2000.

[37] S. Kumar, A. Jantsch, M. Millberg, J. Oberg, J. Soininen, M. Forsell, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, page 117, 2002.

[38] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of Design Automation Conf. (DAC)*, pages 683–689, 2001.

[39] U. Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu. Voltage-frequency island partitioning for GALS-based networks-on-chip. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 110–115, 4-8 Jun 2007.

[40] P. P. Pande, C. Grecu, A. Ivanov, and Saleh R. Design of a switch for network on chip applications. In *Proceedings of International Symposium of Circuits and Systems (ISCAS)*, volume 5, pages 217–220, May 2003.

[41] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Transactions on Computers*, 54(8):1025–1040, August 2005.

[42] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004. San Francisco, CA, USA.

[43] J. Hu and R. Marculescu. DyAD - smart routing for network-on-chip. In *Proceedings of Design and Automation Conference*, pages 260–263. ACM Press, 2004.

[44] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26:62–67, 1993.

[45] G. Chiu. The odd-even turn model for adaptive routing. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):729–738, July 2000.

[46] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low power CMOS digital design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1992.

[47] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Annual international Conference on Mobile Computing and Networking*, pages 251–259, July 2001.

[48] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publisher, 2004.

[49] F. Angiolini, P. Meloni, S. M. Carta, L. Raffo, and L. Benini. A layout-aware analysis of networks-on-chip and traditional interconnects for MPSoCs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 26(3):421–434, 2007.

[50] J. Han and Q. Li. Dynamic power-aware scheduling algorithms for real-time task sets with fault tolerance in parallel and distributed computing environment. In *International Parallel and Distributed Processing Symposium*, pages 6–16, 2005.

[51] C. Piguet, C. Schuster, and J. Nagel. *Static and Dynamic Power Reduction by Architecture Selection*, volume 4148. Springer Berlin / Heidelberg, 2006.

[52] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini. Application-specific power-aware workload allocation for voltage scalable MPSoC platforms. In *International Conference on Computer Design*, pages 87–93, October, 2005 2005.

[53] H. Yoo, K. Lee, and J. K. Kim. *Low-Power NoC for High-Performance SoC Design*. CRC Press, 2008. ISBN 1420051725, 9781420051728.

[54] L. Benini and G. De-Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Springer, 1997.

[55] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Power Driven Microarchitecture Workshop, (ISCA, '98)*, pages 107–112, June 1998.

[56] R. D. Schrimpf and D. M. Fleetwood. *Radiation Effects and Soft Errors in Integrated Circuits and Electronic Device*. World Scientific, 2004.

[57] P. Gil, H. Madeira, and J. Arlat. Fault representativeness. Technical report, LAAS-CNRS (France), 2002.

[58] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. EMAX: A high level error model automatic extractor. In *Proceedings of 9th AIAA Conference on Computing in Aerospace*, pages 1297–1306, 1993.

[59] J. Gracia, J.C. Baraza, D. Gil, and P.J. Gil. Comparison and application of different VHDL-based fault injection techniques. In *Proceedings of the IEEE DFT'01*, pages 233–241, San Francisco, CA, USA, 2001.

[60] A. Fin and F. Fummi. *Languages for System Specification, Part-II: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification from FDL,'03, Grimm, C. (Ed.)*, chapter LAERTE++: An Object Oriented High-Level TPG for SystemC Designs. Springer US, 2004.

[61] M. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *Proceedings*

of the 15th International Symposium on High-Performance Computer Architecture, pages 105–116, Feb. 2009.

[62] A. Ejlali, B. M. Al-Hashimi, M. T. Schmitz, P. Rosinger, and S. G. Miremadi. Combined time and information redundancy for SEU-tolerance in energy efficient real-time systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(4):323–335, April 2006.

[63] R. Melhem, D. Mosse, and E. Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Transactions on Computers*, 53(2):217–231, February 2004.

[64] D. Gil, J. C. Baraza, J. V. Busquets, and P. J. Gil. Fault injection into VHDL models: Analysis of the error syndrome of a microcomputer system. In *In Proceedings of 24 th. EUROMICRO Conference*, volume 1, page 10418, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[65] Soft error in Electronic Memory - A White Paper. Tezzaron Semi., Jan. 2004. http://www.tezzaron.com/about/papers/.

[66] B. M. Al-Hashimi, editor. *System-on-Chip: Next Generation Electronics*. IEE Press, May 2006. Ch: 17.

[67] Synposys Inc. Designware system-level library. http://www.synopsys.com/Tools/SLD/VirtualPlatforms/Pages/SLLibrary.aspx, (last accessed on 23/04/2010).

[68] SystemC. Open SystemC Initiative. http://www.systemc.org.

[69] OSCI. IEEE standard SystemC language reference manual. http://www.systemc.org/downloads/standards/.

[70] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentell. A case study on modeling shared memory access effects during performance analysis of HW/SW systems. In *Proceedings of International Workshop on Hardware-Software Codesign*, pages 117–121, March 1998.

[71] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. Cosyn: hardware-software co-synthesis of embedded systems. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 703–708, New York, NY, USA, 1997. ACM.

[72] R.K. Gupta and G De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, Sept. 1993.

[73] G. Ascia, V. Catania, and M. Palesi. An evolutionary approach to network-on-chip mapping problem. In *IEEE Congress on Evolutionary Computation, 2005*, volume 1, pages 112–119, 2-5 Sept 2005.

[74] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 864–869, Washington, DC, USA, 2005. IEEE Computer Society.

[75] T. Kogel, R. Leupers, and H. Meyr, editors. *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms*, chapter System Level Design Principles, pages 33–42. Springer, 2006.

[76] F. Balarin, P. D. Giusto, A. Jurecska, M. Chiodo, C. Passerone, A. Sangiovanni-Vincentelli H. Hsieh, E. Sentovich, B. Tabbara, L. Lavagno, and K. Suzuk. *Hardware-software co-design of embedded systems: the POLIS approach*. Springer, 1997.

[77] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th Conference on Design Automation*. ACM Press, NY, USA, 2001.

[78] T. N. Theis. The future of interconnection technology. *IBM Journal of Research and Development*, 44(3):379–390, May 2000.

[79] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Design, Automation and Test in Europe (DATE), IEEE*, pages 250–256, 2000.

[80] I. Cidon and I. Keidar. Zooming in on network-on-chip architectures. Technical report, ccit 565, Technion Department of Electrical Engineering, 2005.

[81] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. berg, M. Millberg, and D. Lindqvist. Network on chip: an architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, November 2000. Turku, Finland.

[82] K. Goossens, J. Dielissen, and A. Radulescu. ÆTHEREAL network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414– 421, Sept.-Oct. 2005.

[83] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the NOSTRUM network-on-chip. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE), IEEE*, pages 890–895, 2004.

[84] M. Millberg, R. Thid, S. Kumar, and A. Jantsch. The NOSTRUM backbone - a communication protocol stack for networks on chip. In *In Proc. Intl Conference on VLSI Design*, pages 693–696, 2004.

[85] J. Bainbridge and S. Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.

[86] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark (DTU), 2005.

[87] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xPipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 536, Washington, DC, USA, 2003. IEEE Computer Society.

[88] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 114–122, 2008.

[89] A. Maheshwari, W. Burleson, and R. Tessier. Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):299–311, March 2004.

[90] N. Seifert, D. Moyer, N. Leland, and R. Hokinson. Historical trend in alpha-particle induced soft error rates of the alpha microprocessor. In *IEEE International Reliability Physics Symposium*, pages 259–265, Florida, USA, 2001.

[91] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 35–40, 2004.

[92] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt. Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs. In *Proc. 15th IEEE International On-Line Testing Symposium IOLTS 2009*, pages 101–106, 24–26 June 2009.

[93] H. Beitollahi, S. G. Miremadi, and G. Deconinck. Fault-tolerant earliest-deadline-first scheduling algorithm. In *Proc. IEEE International Parallel and Distributed Processing Symposium IPDPS 2007*, pages 1–6, 26–30 March 2007.

[94] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proc. 17th IEEE VLSI Test Symposium*, pages 86–94, 25–29 April 1999.

[95] Y. Cai, S. M. Reddy, and B. M. Al-Hashimi. Reducing the energy consumption in fault-tolerant distributed embedded systems with time constraint. In *8th International Symposium on Quality Electronic Design (ISQED)*, pages 368–373, 2007.

[96] Y. Zhang and K. Chakrabarty. Dynamic adaptation for fault tolerance and power management in embedded real-time systems. *ACM Transactions on Embedded Computing Systems*, 3(2):336–360, May 2004.

[97] Y. Zhang and K. Chakrabarty. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(1):111–125, January 2006.

[98] P. Pop, V. Izosimov, P. Eles, and Zebo Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions of Very Large Integration Circuits*, 17(3):389–402, March 2009.

[99] P. Pop, K. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proceedings of Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 233–238, 2007.

[100] K. Mihic, T. Simunic, and G. De Micheli. Reliability and power management of integrated systems. In *Proc. Euromicro Symposium on Digital System Design DSD 2004*, pages 5–11, 31 Aug.–3 Sept. 2004.

[101] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, pages 66–75, Austin, Texas (USA), June 1994.

[102] J.C. Baraza, Gracia J., D. Gil, and P.J. Gil. Improvement of fault injection techniques based on VHDL code modification. In *Proceedings of Tenth IEEE International High-Level Design Validation and Test Workshop*, pages 19–26, 2005.

[103] T. A. Delong, B. W. Johnson, and A. Joseph P. Iii. A fault injection technique for VHDL behavioral-level models. *IEEE Design and Test*, 13(4):24–33, 1996.

[104] V. Sieh, O. Tschäche, and F. Balbach. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 32, Washington, DC, USA, 1997. IEEE Computer Society.

[105] H.R. Zarandi, S.G. Miremadi, and A. Ejlali. Fault injection into Verilog models for dependability evaluation of digital systems. In *Parallel and Distributed Computing,*

2003. Proceedings. Second International Symposium on, pages 281 – 287, October 2003.

[106] K. K. Goswami, R. K. Iyer, and L. Young. DEPEND: A simulation-based environment for system level dependability analysis. *IEEE Trans. Comput.*, 46(1):60–74, 1997.

[107] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. *Dependable Systems and Networks, International Conference on*, 0:667–668, 2003.

[108] H. Madeira, M. Z. Rela, F. Moreira, and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *EDCC-1: Proceedings of the First European Dependable Computing Conference on Dependable Computing*, pages 199–216, London, UK, 1994. Springer-Verlag.

[109] K. Rothbart, U. Neffe, Ch. Steger, R. Weiss, E. Rieger, and A. Muehlberger. Power consumption profile analysis for security attack simulation in smart cards at high abstraction level. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 214–217, 2005.

[110] S. Misera, H.T. Vierhaus, and A. Sieber. Fault injection techniques and their accelerated simulation in SystemC. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007. DSD 2007.*, pages 587–595, Aug. 2007.

[111] A. Castelnuovot, A. Fin, F. Fummi, and F. Sforza. Emulation-based design errors identification. In *Proceedings of the IEEE DFT'02*, pages 365–371, 2002.

[112] A. Benso and P. Prinetto, editors. *Frontiers in Electronic Testing: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, volume 23. Springer, 2003. ISBN: 978-1-4020-7589-6.

[113] K. Chang, Y. Wang, C. Hsu, K. Leu, and Y. Chen. System-bus fault injection framework in SystemC design platform. In *SSIRI '08: Proceedings of the 2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 211–212, Washington, DC, USA, 2008. IEEE Computer Society.

[114] K. Chang and Y. Chen. System-level fault injection in SystemC design platform. In *Proceedings of 8th International Symposium on Advanced Intelligent Systems (ISIS 2007)*, 2007.

[115] A. Fin, F. Fummi, and G. Pravadelli. *SystemC*, chapter SystemC as a Complete Design and Validation Environment, pages 127–156. Springer US, 2007.

[116] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger. High level fault injection for attack simulation in smart cards. In *Proceedings of the ATS'04*, pages 118– 121, Nov. 2004.

[117] D. Lee and J. Na. A novel simulation fault injection method for dependability analysis. *IEEE Design and Test of Computers*, 26:50–61, 2009.

[118] S. Misera, H. T. Vierhaus, L. Breitenfeld, and A. Sieber. A mixed language fault simulation of VHDL and SystemC. In *Proceedings of 9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 275–279, 2006.

[119] P. Vanhauwaert, R. Leveugle, and P. Roche. A flexible SoPC-based fault injection environment. In *IEEE Design and Diagnostics of Electronic Circuits and systems*, pages 190–195, 2006.

[120] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Limited, UK, second edition, 2006.

[121] A. Sanyal and S. Kundu. On derating soft error probability based on strength filtering. In *13th International On-Line Testing Symposium (IOLTS 2007)*, pages 152–160, Greece, 8-11 July 2007.

[122] M. L. Bushnell and V. D. Agarwal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI*. Springer, 2000. ISBN 0792379918, 9780792379911.

[123] A. Ejlali, B M. Al-Hashimi, P. Rosinger, and S. G. Miremadi. Joint consideration of fault-tolerance, energy-efficiency and performance in on-chip network. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1647–1652, 2007.

[124] ISO/IEC. 13818-2: 1995 (E): MPEG-2 video decoder specification. http://www.iso.org.

[125] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Schanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jairr, S. Venkataramarr, Y. Hoskote, and N. Borkar. An 80 tile 1.28 TFLOPs network-on-chip in 65 nm CMOS. In *In Proceedings of International Solid State Circuit Conference (ISSCC)*, pages 98–100, 2007.

[126] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987. IEEE Computer Society.

[127] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *IET proceedings. Computers and digital techniques*, 152(4):467–472, July 2005.

[128] J. Chang, W. Kim, Y. Bae, J. H. Han, H. Cho, and H. Jung. Performance analysis for MPEG-4 video codec based on on-chip network. *ETRI Journal, Korea*, 27(5):497–503, October 2005.

[129] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das. Exploring fault-tolerant network-on-chip architectures. In *International Conference on Dependable Systems and Networks (DSN)*, pages 94–104, June 2006.

[130] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, and C. R. Das. Design and analysis of an NoC architecture from performance, reliability and energy perspective. In *Proceedings of the ACM Symposium on Architecture for Networking and Communications Systems*, pages 173–182, 2005.

[131] T. Dumitraş, S. Kerner, and R. Mărculescu. Towards on-chip fault-tolerant communication. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASPDAC)*, pages 225–232, New York, NY, USA, 2003. ACM.

[132] Y. Massoud A. Hosseini, T. Ragheb. A fault-aware dynamic routing algorithm for on-chip networks. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2653–2656, Washington, USA, 2008.

[133] D. Puschini and F. Clermidy. A comparison between NoC and bus architectures based on a real-application. In *Proceedings of ReCoSoC, 2006*, pages 194–200, Montelier, France, July 2006.

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

[135] G. De-Micheli and L. Benini. *Networks on Chips*. Morgan Kaufmann, first edition, 2006.

[136] L. Benini. Application specific NoC design. In *Proceedings of the conference on Design, automation and test in Europe*, pages 491–495, 2006.

[137] J. Xu, W. Wolf, J. Henkel, and S. T. Chakradhar. A design methodology for application-specific networks-on-chip. *ACM Transactions on Embedded Computing Systems*, 5(2):263–280, 2006.

[138] A. Mihal and K. Keutzer. *Mapping Concurrent Applications onto Architectural Platforms, Jantsch, A. and Tenhunen, H. (Ed.s)*. Kluwer Academic Publishers, 2003. ch: 3, 39–59.

[139] S. Vassiliadis and I. Sourdis. FLUX interconnection networks on demand. *The EUROMICRO Journal of Systems Architecture*, 53(10):777–793, Oct. 2007.

[140] F. Xiao, D. Li, G. Du, Y. Song, D. Zhang, and M. Gao. Design of AXI bus based MPSoC on FPGA. In *ASID'09: Proceedings of the 3rd international conference on*

Anti-Counterfeiting, security, and identification in communication, pages 560–564, Piscataway, NJ, USA, 2009. IEEE Press.

[141] J. L. Mitchell, C. Fogg, D. J. LeGall, and W. B. Pennebaker. *MPEG Video Compression Standard*. Springer, 1997.

[142] S. Osborne, A.T. Erdogan, T. Arslan, and D. Robinson. Bus encoding architecture for low-power implementation of an amba-based soc platform. *IEE Proceedings - Computers and Digital Techniques (IEE CDT)*, 149:152–156, 2002.

[143] A. Dalirsani, M. Hosseinabady, and Z. Navabi. An analytical model for reliability evaluation of NoC architectures. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 49–56, Washington, DC, USA, 2007. IEEE Computer Society.

[144] A. Jalabert, S. Murali, L. Benini, and G. De-Micheli. xpipesCompiler: A tool for instantiating application specific networks on chip. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20884, Washington, DC, USA, 2004. IEEE Computer Society.

[145] P. Cherriman and L. Hanzo. Error-rate-based power-controlled multimode H.263-assisted videotelephony. *IEEE Transactions on Vehicular Technology*, 48(5):1726–1738, Sep. 1999.

[146] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM SIGOPS Operating Systems Review*, 35(5):89–102, Dec. 2001.

[147] A. Mohsen and R. Hofmann. *Integrated Circuit and System Design: Power and Timing Modeling, Optimization and Simulation, Vassilis Paliouras, Johan Vounckx, Diederik Verkest (Ed.s)*, chapter Power-Aware Scheduling for Hard Real-Time Embedded Systems using Voltage Scaling Enabled Architectures, pages 127–136. Birkhuser, 2005.

[148] V. Chandra and R. Aitken. Impact of voltage scaling on nanoscale SRAM reliability. In *Proc. DATE '09. Design, Automation & Test in Europe Conference & Exhibition*, pages 387–392, 20–24 April 2009.

[149] N. Soundararajan, N. Vijaykrishnan, and A. Sivasubramaniam. Impact of dynamic voltage and frequency scaling on the architectural vulnerability of GALS architectures. In *ISLPED '08: Proceeding of the 13th international symposium on Low power electronics and design*, pages 351–356, New York, NY, USA, 2008. ACM.

[150] P. A. Lee, T. Anderson (Ed. J. C. Laprie, A. Avizienis, and H. Kopetz). *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., 2nd edition, 1990.

[151] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):112–126, Oct. 2001.

[152] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40, 1989.

[153] H. Wang, S. Baldawa, and R. Sangireddy. Dynamic error detection for dependable cache coherency in multicore architectures. In *Proc. 21st International Conference on VLSI Design VLSID 2008*, pages 279–285, 4–8 Jan. 2008.

[154] K. Wu and D. Marculescu. Power-aware soft error hardening via selective voltage scaling. In *Proc. IEEE International Conference on Computer Design ICCD 2008*, pages 301–306, 12–15 Oct. 2008.

[155] B. W. Johnson, editor. *Design & analysis of fault tolerant digital systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[156] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Yuan Xie. Reliability-centric high-level synthesis. In *Proc. Design, Automation and Test in Europe*, pages 1258–1263, March 2005.

[157] W. Heidergott. SEU tolerant device, circuit and processor design. In *Proc. 42nd Design Automation Conference*, pages 5–10, 13–17 June 2005.

[158] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W. L. Hung. Reliability-centric hardware/software co-design. In *Proc. Sixth International Symposium on Quality of Electronic Design ISQED 2005*, pages 375–380, 21–23 March 2005.

[159] S. Mitra, M. Zhang, N. Seifert, T. M. Mak, and K. S. Kim. Soft error resilient system design through error correction. In *Proc. IFIP International Conference on Very Large Scale Integration*, pages 332–337, 16–18 Oct. 2006.

[160] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. 36th Annual IEEE/ACM International Symposium on MICRO-36 Microarchitecture*, pages 29–40, December 2003.

[161] H. T. Nguyen and Y. Yagil. A systematic approach to SER estimation and solutions. In *Proc. IEEE International 41st Annual Reliability Physics Symposium*, pages 60–70, 30 March–4 April 2003.

[162] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proc. International Conference on Dependable Systems and Networks*, pages 61–70, 28 June–1 July 2004.

[163] GOOGLE summer-of-code. Real-time executive for embedded systems. http://www.rtems.com/, (last accessed on 23/04/2010).

[164] D. Atienza, F. Angiolini, S. Muralia, A. Pullini, L. Benini, and G. De-Micheli. Network-on-chip design and synthesis outlook. *Integration-The VLSI journal*, 41(2):340–359, February 2008.

[165] D. Brooks, V. Tiwari, and M. Martonosi. WATTCH: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[166] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE II)*, April 2006.

[167] M.H. Pinson, S. Wolf, and R.B. Stafford. Video performance requirements for tactical video applications. In *IEEE Conference on Technologies for Homeland Security*, pages 85–90, 16-17 May 2007.

[168] S. Gringeri, R. Egorov, K. Shuaib, A. Lewis, and B. Basch. Robust compression and transmission of MPEG-4 video. In *Proceedings of ACM Multimedia*, pages 113–120, Orlando, Florida, USA, 1999.

[169] K. Srinivasan, V. Ramamurthi, and K.S. Chatha. A technique for energy versus quality of service trade-off for MPEG-2 decoder. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design*, pages 313–316, 19-20 Feb 2004.

[170] T. Karnik and P. Hazucha. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, April 2004.

[171] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of International Sympoisum of Low Power Electronics and Design (ISLPED)*, pages 132–137, 2004.

[172] H. Orsilla, T. Kangas, E. Salminen, T.D. Hmlinen, and M. Hnnikinen. Automated memory-aware application distribution for multi-processor system-on-chips. *Journal of Systems Architecture: the EUROMICRO Journal*, 53(11):795–815, 2007.

[173] J.T. Blake, A.L. Reibman, and K.S. Trivedi. Sensitivity analysis of reliability and performability measures for multiprocessor systems. *ACM SIGMETRICS Performance Evaluation Review*, 16(1):177–186, 1988.

[174] P. Malani, P. Mukre, Q. Qiu, and Q. Wu. Adaptive scheduling and voltage scaling for multiprocessor real-time applications with non-deterministic workload. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 652–657, 2008.

[175] M.S. Beg, Y. C. Chang, and T. F. Tang. Performance evaluation of error resilient tools for MPEG-4 video transmission over a mobile channel. In *Proceedings of IEEE International Conference on Personal Wireless Communications*, pages 285–289, 15-17 Dec. 2002.

[176] H. Joshi, S. S. Verma, and G. K. Sharma. *Lecture Notes in Computer Science: Reconfigurable Computing: Architectures and Applications*, volume 3985/2006, chapter Quality Driven Dynamic Low Power Reconfiguration of Handhelds, pages 59–64. Springer Berlin / Heidelberg, 2006.

[177] A. Andrei, P. Eles, Z. Peng, M. T. Schmitz, and B. M. Al-Hashimi. Energy optimization of multiprocessor systems on chip by voltage selection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(3):262–275, March, 2007 2007.

[178] L. Ingber. Adaptive Simulated Annealing. http://alumni.caltech.edu/∼ingber/, 1993, (last accessed on 23/05/2010).

[179] RTRG. Random task and resource graph tool. http://www.zepler.net/∼ras06r/rtrg, (last accessed on 23/04/2010).

[180] W. J. Van Gils. A triple modular redundancy technique providing multiple-bit error protection without using extra redundancy. *IEEE Transactions on Computers*, C-35(7):623–631, July 1986.

[181] J. A. Abraham and D. P. Siewiorek. An algorithm for the accurate reliability evaluation of triple modular redundancy networks. *IEEE Transactions on Computers*, 23(7):682–692, July 1974.

[182] L. Anghel, D. Alexandrescu, and M. Nicolaidis. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In *Proceedings of Intl. Symposium on Integrated Circuit Design and System Design*, page 237, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[183] J. A. Profeta, N. P. Andrianos, Bing Yu, B. W. Johnson, T. A. DeLong, D. Guaspart, and D. Jamsck. Safety-critical systems built with COTS. *Computer*, 29(11):54–60, Nov. 1996.

[184] S. Wang, J. Hu, and S. G. Ziavras. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Transactions on Computers*, 58(9):1171–1184, Sept. 2009.

[185] D. Kwai and B. Parhami. Fault-tolerant processor arrays using space and time redundancy. In *Proc. IEEE Second International Conference on Algorithms and Architectures for Parallel Processing ICAPP '96*, pages 303–310, 11–13 June 1996.

[186] D. J. Soudris, P. Poechmueller, E. D. Kyriakis-Bitzaros, M. K. Birbas, C. E. Goutis, and M. Glesner. Design methodology for systematic derivation of fault-tolerant array processors. In *Proc. CompEuro '92 'Computer Systems and Software Engineering'*, pages 562–567, 4–8 May 1992.

[187] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10918, Washington, DC, USA, 2003. IEEE Computer Society.

[188] G. Li, F. Hu, and L. Yuan. An energy-efficient fault-tolerant scheduling scheme for aperiodic tasks in embedded real-time systems. In *Proc. Third International Conference on Multimedia and Ubiquitous Engineering MUE '09*, pages 369–376, 4–6 June 2009.

[189] B. H. Meyer and D. E. Thomas. Rethinking the synthesis of buses, data mapping, and memory allocation for MPSoC. *Design Automation for Embedded Systems*, Online(1), June 2008.

[190] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, April 2005.

[191] S. Ghosh, S. Basu, and N. A. Touba. Joint minimization of power and area in scan testing by scan cell reordering. In *ISVLSI '03: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, page 246, Washington, DC, USA, 2003. IEEE Computer Society.

[192] T. Karnik, J. Tschanz, B. Bloechel, P. Hazucha, P. Armstrong, S. Narendra, A. Keshavarzi, K. Soumyanath, G. Dermer, J. Maiz, S. Borkar, and V. De. Impact of body bias on alpha- and neutron-induced soft error rates of flip-flops. In *Digest of Technical Papers. 2004 Symposium on VLSI Circuits*, pages 324–325, 2004.

[193] J. Ahmed and C. Chakrabarti. A dynamic task scheduling algorithm for battery powered DVS systems. In *Proc. International Symposium on Circuits and Systems ISCAS '04*, volume 2, pages II–813–16, 23–26 May 2004.

[194] ARM. ARM7TDMI: Low power processor. http://www.arm.com/products/CPUs/ARM7TDMI.html, (last accessed on 23/04/2010).