High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
M. Sc. Zheng Wang
aus Tianjin, China

Berichter: Universitätsprofessor Dr.-Ing. Anupam Chattopadhyay
Universitätsprofessor Dr.-Ing. Tobias G. Noll

Tag der mündlichen Prüfung: 02.12.2015

Diese Dissertation ist auf den Internetseiten
der Hochschulbibliothek online verfügbar.

Acknowledgements

This thesis is the result of my work as research assistant at the Institute for Communication Technologies and Embedded Systems (ICE) at the RWTH Aachen University. During this time I have been accompanied and supported by many people. It is my great pleasure to take this opportunity to thank them.

My most sincere thanks go to my advisors, Prof. Anupam Chattopadhyay and Prof. Tobias Noll. Anupam has been extremely helpful and tremendously inspiring throughout my PhD study. Prof. Noll has been impressively knowledgeable while patient with my ideas and mistakes. Their thoughtful advices have greatly contributed to this work and influenced me for my future career.

Special thanks go to my defense committee members, Prof. Andrei Vescan and Prof. Renato Negra for spending their time, offering oral exam, giving me feedback, and attending my defense session.

Several colleagues in ICE and EECS have assisted and encouraged me during the past five years for my work and personal life. Among them I would like to show my deep appreciation to Ayesha Khalid, Zoltán Rákossy and Michael Meixner. Furthermore, I would like to thank my students Xiao Wang, Chao Chen, Lai Wang, Renlin Li, Hui Xie, Liu Yang, Saumitra Chafekar, Alessandro Littarru, Shazia Kanwal, Kolawole Soretire, Emmanuel Ugwu, Dan Yue, Kapil Singh, Piyush Sharma and Sai Rama Usha Ayyagari for their consistent contribution.

I would also like to thank my family: My parents and parents-in-law for supporting me spiritually throughout writing this thesis and my life in general. And finally, infinite gratitude to my beloved wife as well as my son.

Zheng Wang, in February 2016

Abstract

Continuous technology scaling in semiconductor industry forces reliability as a serious design concern in the era of nanoscale computing. Traditional device and circuit level reliability estimation and error mitigation techniques neither address the huge design complexity of modern system nor consider architecture and system-level error masking properties. An alternative approach is to accept and expose the unreliability to all layers of computing and possibly mitigate the errors with device-, circuit-, architectural or software techniques. To enable cross-layer exploration of reliability against other performance constraints, it is essential to accurately model the errors in nanoscale technology and develop a smooth tool-flow at high-level design abstractions so that error effects can be estimated, which assists the development of high-level fault-tolerant techniques. In this dissertation, a high-level reliability estimation and exploration framework for MPSoC is developed.

To estimate reliability at early design stages, a high-level fault simulation framework is constructed for generic architecture models and integrated into a commercial processor design environment. The fault injector is further extended for system-level modules. A power/thermal/timing error co-simulation framework is demonstrated for integrating fault injection with simulation of physical properties. To further speed up reliability estimation, an analytical method is proposed to calculate vulnerability of individual logic blocks, from which application level error probabilities are deduced. A formal technique is introduced to predict error effects by tracking error propagation. Finally, design diversity metric is utilized to quantify the robustness of redundancy in system-level computing elements.

The contributions in reliability exploration include several novel architectural fault-tolerant techniques. Opportunistic redundancy detects errors by re-executing the instructions only if there are underutilized resources. Asymmetric redundancy unequally protects memory elements based on criticality analysis of data and instructions. Error confinement replaces any erroneous result with the best available estimate from statistical characteristics. For system-level fault tolerance, a core reliability-aware task mapping algorithm is demonstrated on a heterogeneous multiprocessor platform. A theoretical approach to construct ad-hoc fault tolerant network for arbi-

trary task graph with optimal amount of connecting edges is elaborated and verified by exhaustive search based algorithm.

The methodologies proposed in this dissertation are going to be critical for future semiconductor technology nodes, where reliability is going to be a permanent problem. Further research directions are outlined to take this research forward.

Contents

1	Introduction	1
1.1	Contribution	2
1.2	Outline	4
2	Background	5
2.1	Reliability Definition	5
2.2	Fault, Error and Failure	5
2.3	Hardware Faults	6
2.3.1	Origins	6
2.3.2	Fault Models	8
2.4	Soft Error	8
2.4.1	Evaluation Metrics	8
2.4.2	Scaling Trends	9
3	Related Work	11
3.1	Fault Injection and Simulation	11
3.1.1	Physical Fault Injection	11
3.1.2	Simulated Fault Injection	12
3.1.3	Emulated Fault Injection	15
3.2	Analytical Reliability Estimation	16
3.2.1	Architecture Vulnerability Factor Analysis	16
3.2.2	Probablistic Transfer Matrix	17
3.2.3	Design Diversity Estimation	18
3.3	Architectural Fault-tolerant Techniques	19
3.3.1	Traditional Fault-tolerant Techniques	19

3.3.2	Approximate Computing	22
3.4	System-level Fault Tolerant Techniques	26
3.4.1	Reliability-aware Task Mapping	26
3.4.2	Fault-tolerant Network Design	27
4	High-level Fault Injection and Simulation	29
4.1	Architectural Fault Injection	29
4.1.1	Methodologies	30
4.1.2	LISA-based Fault Injection	32
4.1.3	Timing Fault Injection	37
4.1.4	Experimental Results	39
4.1.5	Summary	44
4.2	System-level Fault Injection	44
4.2.1	Fault injection for system modules	45
4.2.2	Experimental results	46
4.2.3	Summary	48
4.3	High-level Processor Power/Thermal/Delay Joint Modelling Framework	49
4.3.1	High-level Power Modeling and Estimation	50
4.3.2	LISA-based Thermal Modeling	59
4.3.3	Thermal-aware Delay Simulation	63
4.3.4	Automation Flow and Overhead Analysis	67
4.3.5	Summary	69
5	Architectural Reliability Estimation	71
5.1	Analytical Reliability Estimation Technique	71
5.1.1	Operation Reliability Model	72
5.1.2	Instruction Error Rate	73
5.1.3	Application Error Rate	74
5.1.4	Analytical Reliability Estimation for RISC Processor	75
5.1.5	Summary	77
5.2	Probabilistic Error Masking Matrix	78
5.2.1	Logic Masking in Digital Circuits	80
5.2.2	PeMM for Processor Building Blocks	81

5.2.3	PeMM Characterization	84
5.2.4	Approximate Error Prediction Framework	86
5.2.5	Results in Error Prediction	89
5.2.6	Summary	93
5.3	Reliability Estimation using Design Diversity	94
5.3.1	Design Diversity	95
5.3.2	Graph-based Diversity Analysis	97
5.3.3	Results in Diversity Estimation	103
5.3.4	Summary	107
6	Architectural Reliability Exploration	109
6.1	Opportunistic Redundancy	109
6.1.1	Opportunistic Protection	110
6.1.2	Implementation	112
6.1.3	Experimental Results	117
6.1.4	Summary	121
6.2	Processor Design with Asymmetric Reliability	122
6.2.1	Asymmetric Reliability	123
6.2.2	Asymmetric Reliability Exploration	126
6.2.3	Summary	133
6.3	Approximate Computing with Statistical Error Confinement	134
6.3.1	Proposed Error Confinement Method	135
6.3.2	Realizing the Proposed Error Confinement in a RISC Processor .	136
6.3.3	Case Study and Statistical Analysis	138
6.3.4	Results	140
6.3.5	Summary	144
7	System-level Reliability Exploration	147
7.1	System-level Reliability Exploration Framework	147
7.1.1	Platform and Task Manager Firmware	148
7.1.2	Core Reliability Aware Task Mapping	151
7.1.3	Experimental Results	153
7.1.4	Summary	156

7.2	Reliable System-level Design using Node Fault Tolerance	156
7.2.1	Node Fault Tolerance in Graph	157
7.2.2	Construct NFT for Arbitrary Graph	158
7.2.3	Verify NFT Graphs using Task Mapping	160
7.2.4	Experiments for Node Fault Tolerance	163
7.2.5	Summary	164
8	Conclusion and Outlook	167
8.1	Conclusion	167
8.2	Outlook	168
	Glossary	171
	List of Figures	175
	List of Tables	179
	Bibliography	181

Chapter 1

Introduction

The last few decades have witnessed continuous scaling of CMOS technology, guided by Moore's Law [133], to support devices with higher speed, less area and less power. Though there have been varying arguments on how long the scaling can be continued, it is undisputed that there is a reach of classical physics on supporting deterministic circuit behavior, which is limited by the thickness of an atom. The current sub-micron CMOS technology generation is already facing several challenges, resulting in a broad class of problems known as reliability. According to International Technology Roadmap for Semiconductors (ITRS) [11], reliability and resilience across all design layers constitute a long-term grand challenge.

Reliability is influenced by several trends. First, soft errors caused by external radiation are increasingly reported, even at ground conditions [53]. Second, increasing power dissipation leads to thermal stress which affects design lifetime as well as soft error rates [37]. Third, continuous technology scaling gives rise to increased permanent errors caused by process variation [183]. Fourth, frequency and voltage over-scaling targeting timing margin exploration save power and performance budgets but also introduce timing errors [57]. Finally, new kinds of innovative fault-based attacks against cryptographic modules [175] make fault-tolerant design important. Besides, fault tolerance is always mandatory for safety-critical application domains such as aerospace, biomedical, automotive and infrastructure.

The effects of reliability challenges can only be accurately modelled at low levels of design abstractions. For instance, at device level the vulnerability of a transistor against striking particles is analysed according to its physical properties. The deviation on the threshold voltage caused by process variation and temperature shift is evaluated by diffusion effects of chemical elements. At circuit level the generation, propagation and attenuation of the transient current pulse are simulated using SPICE. However, despite its accuracy, low level reliability analysis and simulation are extremely time consuming which can not address the huge design complexity of modern computing system with hundreds of computing elements.

Furthermore, error mitigation techniques at low levels ignore the architectural and application-level error masking abilities which result in conservative design choices affecting performance. An alternative approach is to accept and expose the unreliability to all the layers of computing and mitigate the error effects with high-level techniques. For example, an aggressive voltage scaling of the device may lead to higher runtime performance at the cost of timing errors, which can be corrected by architectural techniques [57]. An uncorrected data error representing the color of a pixel in an image can be intrinsically tolerated by the limit of human perception [17].

Figure 1.1: Overall flow of high-level reliability estimation and exploration

1.1 Contribution

A key ingredient of successful cross-layer exploration of reliability against other performance constraints (e.g. power, temperature, speed) is to accurately model the errors in nanoscale technology and develop a smooth tool-flow at high-level design layers to estimate error effects, which assists the development of high-level fault-tolerant techniques. In this thesis, multiple challenges for developing the reliability-estimation and exploration framework are tackled. Figure 1.1 shows the overall flow with detailed discussion on individual blocks in the following.

- **High-level Fault Injection and Simulation**

Fault injection, which is an important setup for reliability exploration, is discussed in Chapter 4. Section 4.1 presents the fault injection tool for generic cycle-accurate architecture models which has been integrated into commercial processor design framework. The faults can be injected at both combinational logic and memory cells while achieve similar accuracy as state-of-the-art RTL fault injection. Two modes of fault injection are supported. In the configurable mode, faults are defined based on user's configuration through graphical interface. In the timing mode, logic delay faults are injected based on the statistics from low level timing analysis and variation function. In Section 4.2 the fault injector is extended for system-level modules described in SystemC language. An interesting case study is to relate delay faults injection with power consumption and runtime temperature variation, while the co-simulation framework for power, temperature and delay faults is proposed in Section 4.3.

- **High-level Reliability Estimation**

Architectural reliability can be fast estimated through analytical methods, which are discussed in Chapter 5. Section 5.1 presents an analytical estimation technique based on graph representation of processor architecture. The vulnerability and logic masking capability of vertexes in the graph representing logic blocks can be fast characterized. The edges in the graph which link the vertexes direct the estimation of instruction and application-level error probability. In Section 5.2 such analytical method is further extended as a formal algorithmic approach to predict error effects by tracking error propagation and attenuation in a graph network representing dynamic processor behavior. A different reliability estimation technique is proposed in Section 5.3 to quantify the robustness of a redundant system against common mode failure using design diversity. Assisted by a graph indicating exclusiveness information of architecture modules, the approach quantifies the potential of fault tolerance for different computing elements using MTTF metric.

- **Architectural Reliability Exploration**

Three novel architectural fault-tolerant techniques are proposed in Chapter 6. The first technique, named as opportunistic redundancy in Section 6.1, introduces a passive error detection policy for algorithmic units by re-executing the instruction only if there exists underutilized resources, which incurs very small performance penalty. The approach is benchmarked with aggressive policy where all instructions are double executed to verify the correctness of results. The second technique, named as asymmetric redundancy in Section 6.2, presents an unequal error protection technique for storage elements based on criticality analysis. Different schemes of asymmetric protection are investigated for instruction and data words, with static or dynamic criticality assignment. The last technique, named as error confinement in Section 6.3, exploits the statistical characteristics of any target application and replaces any erroneous data in memory with the best available approximation of that data rather than correcting every single error. All techniques are demonstrated on embedded processors with customized architecture extension.

- **System-level Reliability Exploration**

In Chapter 7 fault tolerant techniques in system-level design are presented which focus on reliability-aware task mapping and reliable network design. Section 7.1 introduces a heuristic task mapping algorithm which jointly considers task reliability requirement and core reliability level. The mapping technique is demonstrated on a heterogeneous multiprocessor platform with customized firmware layer for fault injection, topology exploration and task management. Section 7.2 presents a theoretical approach to construct ad-hoc fault tolerant network for arbitrary task graph, which contains an optimal amount of connecting edges. Exhaustive search based graph verification algorithm is demonstrated and real world tasks are applied to show the generic feature of proposed technique.

1.2 Outline

The dissertation is organized as following. Chapter 2 presents the background of recent reliability issues. Chapter 3 provides a summary on the related work of reliability estimation and exploration. Chapter 4 describes the fault injection framework which targets both architectural and system-level design. Chapter 5 elaborates several reliability estimation techniques for architecture components. Chapter 6 concentrates on different fault tolerant techniques for error resilience in architecture level. Chapter 7 illustrates proposed system-level techniques enhancing reliability. The conclusion and outlook of this dissertation are presented in Chapter 8.

Chapter 2

Background

In this chapter, fundamental knowledge on reliability are discussed, including reliability definition, fault classification and fault models. In the next soft error and its evaluation metrics are elaborated, which is heavily used in the following chapters.

2.1 Reliability Definition

Reliability is in a broad sense one attribute of *dependability*, which describes the ability of the system to deliver its intended service [51]. Reliability measures the capability of *continuous* delivery of *correct* service. Formally, reliability $R(t)$ at time instance t defines the probability that system performs without failure in time range $[0, t]$, provided that system functions correctly at time 0. Reliability is a function of time, where longer time will reduce the system reliability. Another attribute of dependabilities is *availability*. Availability $A(t)$ defines the probability that system performs correctly at time t , which is often used when occurrence of failures is tolerated. For instance, system down time per year in network application is a measure of availability, since short failure time in network is allowed by the users.

2.2 Fault, Error and Failure

The definition of reliability shows its strong relationship with failure, which indicates the occurrence of unexpected behavior of a *system*. The definition of failure differs with the scope of the system. In a software system, the failure can be defined as a wrong value in the program outputs. In a hardware system such as the architecture of a processor, the failure can be interpreted as a mismatch value of the values stored into memories. Generally, failure is strong correlated with the system under discussion.

Error is a wrong value during *computation*, which is the cause of failure. For instance, error can be viewed from architecture perspective as a logic value which differs the state of the circuits from the correct one. Explicitly, an error occurs when the sequential logic of the circuits exhibits an unexpected value. The sequential logic includes register file and pipeline registers. Not all errors lead to failure. For instance, the erroneous values in register file is overwritten before stored into the data memory. An error in the pipeline register can be ignored when the computation never uses such operand. Generally, errors can result in different effects, such as benign fault, Silent Data Corruption (SDC), Detected Unrecoverable Error (DUE) and system crash. The author in [204] illustrates various system-level effects of error.

Fault from hardware perspective is the *physical* defect or temporal malfunctions, which is the cause of error. Fault can be also defined from software perspective such as a bug in the program due to incorrect specification or human mistakes. The dissertation concentrate on hardware related faults. Not all faults can result in errors. Generally, four masking mechanisms prevent the faults in outputs of combinatorial gates from forming errors in the storages:

- **Electrical Masking:** The fault in the form of current pulse attenuates its electrical strength during the propagation through logic network. The duration of the pulse increases while the amplitude decreases. When the pulse reaches the sequential logic, the attenuated amplitude may not be strong enough to be launched into the storage cell. A technique to model electrical masking is presented in [137].
- **Logic Masking:** Combinatorial logic has its intrinsic masking ability. For instance an 2-to-1 AND gate which has one input of value zero, will mask the fault on the other input.
- **Timing Masking:** The faulty current pulse propagates to the input of sequential logic with enough strength. However, it can not be latched into the flip-flop since it does not arrive within the timing window for data latching. The timing window is the sum of setup and hold time of the flip-flop [8].

2.3 Hardware Faults

2.3.1 Origins

2.3.1.1 Transient Fault

Transient fault, which are often named as soft fault or glitches, is temporal hardware fault which keeps active for a limited time duration. Transient fault is no longer present when its driving source disappears. The causes of radioactive related transient faults can be alpha particles, cosmic arrays and thermal neutrons. When such particles strike the transistors, electron-hole pairs are formed and collected by the transistor's source and drain area. Once the charges are stronger enough, a current pulse occurs and can potentially flip the value of the memory cell, which resulted in a *Single Event Upset* (SEU) or produce glitches named *Single Event Transient* (SET) to the logic output. The smallest amount of charge to cause the SEU is called *critical charge* Q_{crit} . Higher Q_{crit} will reduce the probability of SEU, however, also reduce the speed of the logic transition for the circuits.

- **Alpha Particle** consists of two protons and two neutrons. They are usually from radioactive nuclei during their decay. The emitters of alpha particles are usually the impurities in the device package, which can potentially affect the active region. As the progress of packaging technologies such as 3D packaging, the

active region has become very close to the solder bumps so that alpha particles with low energy can also cause transient faults.

- **Cosmic Rays** are the main source of transient faults for chips applied in terrestrial domain. Cosmic array is a high energy neutron flux, whose density is mainly determined by altitude and locations. Neutrons are uncharged particles which do not interact with charged electrons or holes. Consequently they are highly penetrating and cause low protection efficiency by shielding. Recently, SEU caused by cosmic array are increasingly reported, even at ground conditions [53].
- **Thermal Neutrons** In contrast to the high energy neutrons from cosmic rays, thermal neutrons are the terrestrial neutron flux from the surrounding environment. Recently, the circuits become sensitive to the thermal neutron flux due to the appliance of boron-based glasses in manufacturing [48].

2.3.1.2 Permanent Fault

Permanent faults refer to the faults which are unrecoverable. For CMOS technology they can be classified as extrinsic and intrinsic faults. Extrinsic faults are caused during device manufacturing by contamination or burn-in testing. Intrinsic faults are directly related to the CMOS ageing effects, where the performance of device degrades through time. Several ageing effects are briefly reviewed as following.

- **Electromigration (EM)** refers to the mechanism that causes void region in metal lines or devices, which prevents the further movement of electrons. Electrons hit the metal atoms during the movement through metal wires. With sufficient momentum of the electrons, the atoms can be displaced in the direction of electron movement. High temperature increases the momentum of electrons which leads to faster displacement of atoms. Such mechanism finally result in a void region in the metal wire.
- **Hot Carrier Injection (HCI)** degrades the maximal operating frequency of the chip. HCI originates from the ionization effect when the electrons in the channel hit the atoms around the drain-substrate interface. The electron-hole pairs with sufficient energy, which are caused during ionization, can potentially enter the oxide to occur damage. Such effect raises the threshold voltage of transistor and reduces the operating frequency by 1% to 10% during the device lifespan of 15 years.
- **Negative Bias Temperature Instability (NBTI)** also degrades operating frequency by increasing the threshold voltage of PMOS transistor. The negative bias under high temperature cause the stress to the PMOS transistor, which results in the breaking of silicon-hydrogen bonds in the oxide interface. The free hydrogen atoms create traps at oxide-channel interface by combining with oxygen or nitrogen atoms. This finally leads to the reduction in holes mobility and negative

shift of PMOS threshold voltage. NBTI is predicted to be the most critical ageing effect for CMOS technology under 45nm technology [21].

2.3.2 Fault Models

To investigate the effects of physical faults on higher level of design abstractions, faults are usually modelled with predefined behaviors. Several prevalent fault models are presented in the next. In practice, the effects of physical fault are modelled using the combination of different fault models below.

- **Stuck-at Fault** is used to model the effect when the memory cells or logic gates permanently stuck at the logic value zero or one. Stuck-at faults are the most common type of fault model.
- **Single Bit-flip Fault** is used to model the transition of logic value to another value. It can be classified as simple bit-flip, where the logic value changes when the fault is injected, and bit-flip within the time window, where the value flips back to its original value after the duration of the fault.
- **Multiple Bit-flip Fault** is used to model the simultaneous change of logic values for multiple bits. It can also model the coupling fault, such as short between multiple logic cells or wires.

2.4 Soft Error

Most of the work in this dissertation focuses on analysis and tolerance of transient faults, which manifest into soft errors. Soft error is a synonym of SEU, which represents the bit-flip of logic value in a memory cell or flip-flop. It results from either the strike of radioactive particles in the memory / flip-flop cell or the latched erroneous value from SET of logic faults. According to the location of errors effected by the fault, SEU can be further classified as Single Bit Upset (SBU), Multiple Bit Upset (MBU) and Multiple Cell Upset (MCU) [92]. Recently, MBU and MCU become important threats for nanoscale technologies [86]. In this section, the evaluation metrics for soft error and its scaling trend are introduced.

2.4.1 Evaluation Metrics

- **Mean-Time-to-Failure (MTTF)** represents the average time between two errors or failures. Assume n components exist in the system, the system MTTF is computed from MTTF from individual component using:

$$MTTF_{sys} = \frac{1}{\sum_{i=1}^n \frac{1}{MTTF_i}} \quad (2.1)$$

Figure 2.1: SER scale trend for SRAM and DRAM [176] Copyright ©2010 IEEE

- **Failure-in-Time (FIT)** FIT with is more favorable than MTTF since it is additive in computation. One FIT indicates an error within 10^9 hours. If the components in the system are independent, the system FIT is the addition of FIT for individual components using:

$$FIT_{sys} = \sum_{i=1}^n FIT_i \quad (2.2)$$

FIT is a typical representation of Soft Error Rate (SER).

2.4.2 Scaling Trends

The drastic reduction of technology size and supply voltage has significant impact on SER of different components. The SER scaling trends for SRAM, DRAM (Figure 2.1a) and b)) and combinatorial logic (Figure 2.2) are presented.

- **SRAM** has a flat decreasing SER trend as technology scales. This is due to the fact that both Q_{crit} of the SRAM cell and the cell area for the particles to strike decrease, which leads to a saturation for the SRAM SER. Figure 2.1b) also shows the SRAM SER per unit area indicating the per chip SER, which is even increasing. Another trend shows the fast increment of MCU, where the ration of MCU to SBU grows from a few percent at 250nm to 50% at 22nm [87]. The work in [64] also investigates the MBU rate for 65nm.
- **DRAM** reduces its SER significantly for new technologies. The reason is that with reduced cell area, the Q_{crit} for DRAM cell remains roughly constant, which makes the particles difficult in upsetting the cell. DRAM vendors achieve this by implementing deeper trenches, more tracks and larger capacitors.

Figure 2.2: SER scale trend for combinational logic [171] Copyright ©2002 IEEE

- Combinatorial logic** Figure 2.2 shows the predicted trend of logic SER rate from Shivakumar [171] from 600nm till 50nm technology, where the logic SER approaches SRAM. The SER is also predicted to increase with running frequency. Such prediction is based on simulation, where recent work in [66] presents that the logic SER is below 30% of nominal latch SER for 32nm fabricated chips.

Chapter 3

Related Work

In this chapter, the related work of this dissertation is elaborated. Initially, fault injection techniques are discussed. Following that, major high-level reliability estimation techniques are briefly illustrated. Afterwards, traditional and state-of-the-art architectural fault tolerant techniques are selectively presented. Finally, several design approaches to enhance system-level reliability are explained.

3.1 Fault Injection and Simulation

Fault injection (FI) has been applied over several decades to validate the device dependability under faulty conditions. The benefits of FI include but are not limited to the following:

- Track the propagation of faults and their consequences in the system.
- Verify the system behavior under tolerated range of faults, which is documented in the device specification.
- Explore efficient fault tolerant techniques in specific faulty environment.
- Estimate fault coverage of testing mechanism in device.
- Understand the behavior of real physical faults and benchmark with high-level fault injection techniques.

Hardware related FI techniques is the focus of this dissertation. According to their implementation mechanism, FI techniques are classified into *physical* FI, *simulated* FI and *emulated* FI. A survey of techniques from individual domain follows in this section.

3.1.1 Physical Fault Injection

Physical FI or hardware FI involves the fault injection using physical sources such as neutron flux or through processor pins. Physical FI can be further classified into contact technique and non-contact technique. The contact technique usually uses pins as the inputs of faults, which can only test selective faults. The non-contact technique involves no direct contact with the source of faults, such as radiation rays, so that the injection location can spread over the device. The physical FI techniques are very fast in speed and able to accurately model low-level faults. The major disadvantages are

the large setup cost, low controllability and observability. Representative physical FI tools are listed in the following.

- *MESSALINE* [9] adopts both active probes and sockets to inject faults through pins of device. It is able to inject multiple fault types including stuck-at, bridging and open faults, while can also control the duration of faults. The injection module can select up to 32 injection points. Test sequences are automatically generated by a manager module, which also performs fault analysis.
- *RIFLE* [121] presents a pin-level FI tool for processor architectures. It is based on the idea of trigger and tracing, which records extensive behavioral information after faults. No feedback circuits are needed for the mismatch detection. RIFLE focuses on its ability for fault analysis, which has been applied to analyse the protection efficiency of multiple fault tolerant modules.
- *FIST* [98] create transient faults to the system using both contact and non-contact techniques. The device is exposed under a radiation environment. Transient faults are created using heavy-ion radiations and injected in random locations. The test system, which includes two computers and the radiation source, is placed inside a vacuum chamber. FIST also supports the injection of power disturbance faults through a MOS transistor locating between the power line and the V_{cc} pin to mimic the power fluctuation.
- *MARS* [63] uses not only heavy-ion radiation but also electromagnetic fields to perform non-contact FI, which is realized by either a chip near a charged probe or a circuit board between two charged plates. MARS also uses dangling wires as antennas to generate the electromagnetic field to test the effect for the pins of device.
- *Van@2011* [190] was proposed recently in the domain of crypto-analysis. This work tries to inject fault through very focused optical beams due to the fact that CMOS transistor is sensitive when facing optical pulse to switch its value. It allows very fine focusing of the optical beam to individual architecture components of the micro-controller.

3.1.2 Simulated Fault Injection

Simulated FI changes the runtime states of the simulator. Compared with physical fault injection, simulated FI does not require a produced chip for testing so that very low cost is incurred. Simulated FI has been adopted heavily in the verification phase of chip. Recently, the emergence of virtual prototyping also shows its usage in system-level design for reliability purpose [198]. Simulated FI achieves maximal controllability and observability due to the available description of architecture. The model under FI can be from multiple design abstractions such as circuit and gate level, register transfer level and system level, where model from higher abstraction implies less controllability for fault injection while faster in simulation speed. In [35]

the author demonstrates the inaccuracy of high-level error injection techniques compared with low level ones, which indicates that the cross layer masking effects play an significant role in fault simulation and analysis.

Generally, simulated fault injection can be classified as *code-modification* (CM) and *simulator commands* (SC) technique. Within the CM category, *Saboteurs* and *Mutant* [91] [18] are the most common ones. While *Saboteurs* add new components to the HDL model, *Mutant* replaces the original model with a modified one. Both of these methods have the limitation that they must modify the source code so that recompilation is needed. The SC technique takes advantage of the simulator commands to modify the signal and variable values of the HDL model during simulation. Such method has the advantage that no recompilation is needed so that extra time caused by fault injection is greatly reduced. The main problem of the SC technique lies in its controllability over the injected places, which means not all of the signals and variables within the architecture can be reached by the simulator commands. Furthermore, the portability among different simulators raises another issue for the SC technique.

In this section, representative simulated FI approaches are shown according to their design abstractions.

3.1.2.1 Gate-level and Register-transfer-level Techniques

Simulated FI techniques for both gate-level and RTL work on the simulation model in either VHDL or Verilog languages, which are discussed together.

- *VERIFY* [172] provides a language extension to VHDL language supporting faults description which enables hardware manufacturers to implement their technology dependent faults as libraries. Multi-threaded fault simulation is applied to increase the speed of fault injection and comparison with golden simulation.
- *MEFISTO-L* [26] uses *Saboteurs* technique to augment the original VHDL module with fault injection capabilities. Automated paring, injection and result extraction blocks are designed to speed up fault simulation. Another tool variation is named *MEFISTO-C* [62], which applies simulator command method to inject fault on the fly. The Vantage Optium VHDL Simulator has been used for parallel simulation on the network of UNIX workstations.
- *GSTF* [18] is an automatic and model independent fault simulation tool which supports main FI techniques such as SC, mutants and saboteurs. A wide range of fault models can be injected. The tool is able to automatically analyse the result from fault configurations in order to validate the fault tolerant mechanisms.
- *FIT* [56] introduces a tool for automatic insertion of hardware redundant and information redundant fault tolerant structures as synthesizable VHDL components and performs fault injection to demonstrate the usability. The designer

provides guidelines for the tool to update the original model. The fault tolerant components are pre-developed as library modules.

- *Berrojo@2002 [20]* describes techniques for speeding-up FI on fault tolerant circuits at RTL. The faults are collapsed with several optimization techniques to reduce the time required for FI.
- *INJECT [212]* is able to inject faults for all design abstraction layers including switch level modules in Verilog modules, which can not be described in VHDL language. Mutants are adopted for fault injection.
- *David@2009 [42]* extends the standard Verilog simulator with fault injection capability through Verilog Programming Interface (VPI). The faults are configured using XML files and scheduled/injected during runtime accordingly. A generic SC based technique for Verilog modules is introduced.

3.1.2.2 System-level Techniques

System-level fault injection techniques works on the simulator models described in high-level languages such as C++ and SystemC. It provides efficient solution for the design of fault tolerant techniques in MPSoC architectures.

- *Chang@2007 [30]* presents the Saboteurs based FI technique for SystemC and demonstrates the usability for different levels of Transaction Level Models (TLM).
- *Misera@2007 [129]* proposes FI techniques for SystemC modules by Saboteurs and Mutant. The work also introduces SC based simulation by extension of SystemC library, which can consequently access the public signals and variables. Several optimization techniques in parallel computing are presented to accelerate the simulation speed. Besides, switch level fault simulation is also presented in the work.
- *Shafik@2008 [169]* proposes a general FI approach for SystemC by replacing the original variable types with FI enabler types. Consequently, the original functions are intact and design modifications are less intrusive. Experiments also show a speed-up in simulation with new data types.
- *Beltrame@2009 [19]* introduces a complete non-intrusive SC-based FI technique for SystemC modules without kernel and module extension. The work is based on the technique named reflective wrapper from the Python language, where a python layer is integrated between SystemC modules and kernels to allow the access of SystemC members and variables. Such elements can be manipulated through command line or parsed from a fault configuration XML file.
- *Lu@2011 [116]* proposes the fault simulation in SystemC by concurrent and comparative simulation (CCS), which was originally applied in functional verification. CCS speeds up simulation by concurrent simulation of many machines

with different fault configurations compared to a reference fault-free one. The module is transformed into a high-level decision diagram, where each node in the diagram is injected with a complex pattern keeping fault free and a set of faulty values. The pattern propagates through the network on all machines to realize parallel simulation. The experiments show a speed-up upto 665x for transient faults.

3.1.3 Emulated Fault Injection

Recently, emulated FI technique has become an active research field due to its faster experiment speed as physical FI, as well as good controllability and observability as simulation technique. Typically, fault injection is implemented on FPGA-based hardware modules through the available HDL codes. It achieves additional benefits in hardware prototyping before the actual deposition of final designs. Selected approaches are presented in the following.

- *FIDYCO* [147] introduces an FI technique in combined hardware/software environment. The hardware side is implemented in FPGA while the software side is in the host machine. Both the design under test and golden node can be implemented in FPGA to speed up FI experiment. The tool provides a flexible and open system for testing further components.
- *FT-UNSHADES* [4] uses the technique of partial reconfiguration from FPGA for FI and capture-feedback mechanism for error observation. Special configuration circuits are used for change values of flip-flops. Bit-flip errors injection are speeded up by direct manipulation of bitstreams.
- *FITVS* [214] demonstrates the library-replace-modelling technique to insert saboteurs in the library modules for FI. Real time emulation is performed without FPGA reconfiguration. Gate-level netlists are manipulated such as the transformation of flip-flop into 8 gates implementation for FI.
- *FuSE* [90] proposes the fault simulation using SEmulator, where both simulation based FI and FPGA accelerated FI can be switched. The integration is transparent so that both fault propagation and huge number of FI experiments are realized simultaneously.
- *FLIPPER* [7] presents the FPGA emulation platform for SEU in the configuration memory. Proton irradiation is performed for FI during ground test. The effects of various protection mechanism are tested in the radiation environment.
- *DFI* [123] is designed for SER estimation of SEUs in memory cells of LEON3 processor cores during emulation. Saboteurs for memory cells and flip-flops are adopted for FI purpose, where the emulation results are instantaneously available in the host PC from Ethernet link port. FI can be performed in single clock cycle when processor runs an application.

- *NETFI* [122] presents a netlist level emulated FI tool, where the FPGA built-in library after FPGA-based logic synthesis is automatically modified to generate netlist with cells capable of SEU and SET injection.
- *Cho@2013* [35] evaluates the accuracy of various FI techniques compared with emulation technique, which injects errors into flip-flops of LEON3 processor. Error checkers are inserted at different design modules to track the error propagation. Based on the experiments of this work, the author further discusses the necessity for conventional FI techniques in [127].

3.2 Analytical Reliability Estimation

Despite of the ability for reliability estimation, fault injection consumes large cost in experiment setup, simulation and system configuration. As an alternative, analytical reliability estimation techniques are proposed to perform fast analysis of system behavior under faults using either statistical data collected from fault simulation or probabilistic analysis of circuits behavior. In this section, three representative analytical reliability estimation techniques are briefly discussed, whose theories are adopted in this dissertation for further proposals.

3.2.1 Architecture Vulnerability Factor Analysis

Architecture Vulnerability Factor (AVF) was proposed in [135] to calculate the probability that a fault within a certain architecture unit (mainly for storage units) will lead to user visible errors. AVF is computed using the processor state bits of *Architecturally Correct Execution* (ACE). A hardware storage contains ACE bits when they are further loaded and processed by instructions which potentially commit values into architectural registers and memories, and un-ACE bits when their values do not affect the following execution of processor. Under pessimistic estimation, the author assumes initially all bits are ACE and removes the ones only if they are shown to be un-ACE. Un-ACE bits can be classified from microarchitectural perspective as idle state bits, mis-speculated state bits, predictor structure bits and Ex-ACE state bits. From architectural perspective, NOP instructions, performance-enhancing instructions, predicated fault instructions and dynamically dead instructions will produce un-ACE bits. The readers are suggested to refer [135] for details of un-ACE bits.

The calculation of ACE bits involves an performance simulator, where performance counters are used to profile and track the instructions. This is demonstrated using Asim framework [54] of IA64 architecture [101] to estimate AVF for instruction queues and execution units. The instruction profiling result, which contains the percentage of committed instructions which contain ACE bits (ACE IPC) and the average cycles of ACE bits' residence time (ACE latency), are provided to the AVF calculation methodology using *Little's Law* [105], which result in architecture and application dependent AVF values.

The generic pessimistic ACE model is further optimized to reduce the non-vulnerable time interval using specific behavior of architecture components, which leads to less conservative techniques for instruction cache [197], data cache [74], L2 cache [31] and register file [132]. However, the author in [65] pointed out that an 6.6x over-estimated error on average is indicated by benchmarking the AVF estimation with ACE analysis and fault injection. Such huge inaccuracy comes from several factors.

- The bitflip fault model assumed originally in ACE analysis is inaccurate with technology scaling, since MBU and MCU are much more prevalent in nanoscale CMOS devices.
- The simple bitflip model is advised to be replaced by biflip with certain probability, since the time instance and location of particle strike directly affect the chances of bitflip.
- Precise classification of ACE bits cannot be made until execution time. Original approach identifies all bits to be ACE unless proved as un-ACE using predefined instructions and architecture states. The potential error defined as the value committed to architecture registers also increases the estimation gap, since most of such errors are later masked due to the program nature itself. It is advised in Section 5.2 [201] of this dissertation that ACE bits can be accurately identified through probabilistic fault tracking analysis in a architectural simulator, which considers fine-grained logic masking effect.

In parallel with architectural ACE analysis, other works take advantage of ACE for analysis of software reliability. [150] proposed compiler optimization techniques to generate reliable code which minimizes the ACE latencies of program variable. The work is further extended in [151] to jointly consider functional correctness and timing reliability. In [204] several software and hardware techniques are proposed to reduce the soft error rate based on fault tracking and ACE analysis.

In summary, despite its fast estimation speed which corresponding to one program run in fault injection technique, ACE analysis incurs significant overestimation which prohibits its application for architectural reliability estimation. Also, ACE analysis can only be applied to storage elements but not combinatorial logic. Another approach for AVF calculation is to perform statistical characterization for architecture components using fault injection, following with the graph-based AVF analysis with AVFs of individual components. Such approach is proposed in Section 5.1 [210].

3.2.2 Probabilistic Transfer Matrix

Krishnaswamy [160] introduced *Probabilistic Transfer Matrix* (PTM) as an circuit-level reliability estimation technique. For a given gate-level circuit, the truth table, which describes the circuit behavior, can be viewed as a matrix contains only zero and one as its elements. The rows of the matrix indicates the binary combination of the inputs of circuits, while the column indices correspond to the outputs. Such matrix is named

as *Ideal Transfer Matrix* (ITM). PTM is obtained from the ITM by allowing its entry element to exhibit real value in the range of $[0, 1]$. Error probability of the circuit is defined to be the deviation of PTM element from its counterpart in ITM. PTM for entire circuits can be derived from PTM of individual gates and connecting wires. To do this, PTM algebra is illustrated which contains operators such as normal matrix product for serial connected circuits, tensor product for parallel connected circuits and swap operator for wire swapping. Additional operators such as *fidelity* is introduced for analyzing logic masking effects for the circuits input with error probabilities. An extension of PTM algebra is also presented in [160] which models the electrical masking effects due to error glitch attenuation through the logic gates [137]. The elements in PTM are replaced using attenuation probability, which is derived from the glitch duration relative to the gate propagation delay.

PTM provides an accurate methodology for error estimation in the outputs of circuits when error probabilistic of specific cells inside the circuit is known as a priori. The approach is accurate compared with ACE based AVF analysis since the derivation comes from low level probabilistic analysis. Although mainly applied for small scale circuits, PTM algebra can handle large circuits under an automated analysis framework. However, PTM suffers from scalability problem for large circuits since the size of the PTM is $2^n \times 2^m$ where n and m imply the total number of *bits* for inputs and outputs. Although optimization techniques are proposed in [159] to compress the size of PTM using algebraic decision diagram (ADD), the derivation of PTM from individual gates is extremely time consuming and impractical. Besides, PTM is applied for handling masking effects in pure hardware, where a processor like architecture needs a joint software and hardware analysis tool for accurate error propagation analysis. Such issue is addressed in Section 5.2 [201] where the dimension of PTM is reduced to $n \times m$ where n and m imply the total number of *signals* for inputs and outputs. The fault propagation is also considered in the simulator with cycle accurate state information of the processor.

3.2.3 Design Diversity Estimation

Redundancy is the fundamental idea for the error detection of fault tolerant system. A redundant system consists of multiple implementations of the same function. Providing the same data as common inputs, the results from each implementation are compared for error detection. A *Common Mode Failure* (CMF) implies the error/failure which can affect each implementation in the same fashion, which is undetectable by the redundant system. Examples of such failure are the power disturbance and electromagnetic coupling, which affects all implementations simultaneously. A redundant system should minimize the chances of CMF. *Design diversity*, which was originally proposed in [13], is used to protect redundant system from CMF by independent generation of two or more hardware/software components. For instance, N-version programming [14] is applied to attain diversity. Hardware diversity is applied in the Primary Flight Computer (PFC) system of Boeing 777 [153] by using processors from different vendors. The principle behind is that the redundant system

with different implementations is prone to have different erroneous outputs when facing errors, which is easier to be detected.

Design diversity is further extended as an quantitative evaluation metric for the redundant system [130], which is defined as a rated average of design diversity for all fault pairs in the system. Design diversity is directly related to system reliability. It is concluded in [130] that for a high rate of CMF, a small quantity of design diversity can significantly increase system reliability. When CMF rate is low, large design diversity is required to improve reliability. Fault injection experiments prove the usage of design diversity as an reliability evaluation metric. Efficient diversity estimation techniques for combinatorial circuits are proposed in [131], which works on circuit structures showing regularity features. For arbitrary circuit, reduction techniques by fault equivalence and fault dominance are adopted to significantly reduce the number of fault pairs for calculation of design diversity.

Compared with ACE and PTM, design diversity is specialized in the analysis of redundant systems which are frequently implemented by spatial redundancy such as Triple Modular Redundancy [117]. Other than a pure theoretical methodology, design diversity needs to be calculated using fault injection experiments, which need to be performed exhaustively for all potential fault pairs in the redundant system. Consequently, design diversity also faces scalability issue for the analysis of modern system. Furthermore, both spatial and temporal redundancy exist in modern processor architecture. To exploit such redundancy, not only circuit level design diversity analysis is needed but also micro-architectural analysis which considers whether redundant components can potentially execute simultaneously. The original quantitative metric is extended into system-level analysis based on activation graph structure of arbitrary processor architectures, which partially addresses the scalability problem of design diversity. The analysis is presented in Section 5.3.1 [202].

3.3 Architectural Fault-tolerant Techniques

In this section prevalent fault tolerant techniques in architecture-level are presented. First, the traditional hardware techniques which ensure the correction of errors once upon their detection are discussed. After that, a recently hot research topic namely approximate computing is investigated, where the reduction in quality-of-service (QoS) can be tolerated for power/energy saving.

3.3.1 Traditional Fault-tolerant Techniques

3.3.1.1 Redundant Execution

Redundant execution involves the techniques to compare the outputs of redundant hardware modules which execute same instruction streams. A mismatch of the compared values triggers the error correction mechanism such as checkpointing [50]. The discussion in the section focuses on the error detection mechanism. Dual-modular Redundancy (DMR) contains the replication of two modules, while the Triple-modular

Redundancy (TMR) involves three redundant threads. In [158] the concept *Sphere of Replication* is introduced to formally define the scale of hardware redundancy, which can be classified as *Lockstepping* and *Redundant Multithreading* (RMT) techniques accordingly. In Lockstepping, a cycle by cycle comparison is performed for each instruction. The redundant hardware copy within the sphere is synchronized with the original one. Every signal from the two copies are compared in each cycle. In contrast, RMT only compares the outputs of *committed* instructions so that the states within each instruction can be different.

Lockstepping provides a large fault coverage for the errors within each implementation. The realization of lockstepping is straightforward since no sophisticated control between two copies are required. However, this comes at the cost of two major drawbacks. First, Lockstepping cause increased amount of CMF errors, since the design diversity of the same implementations are very low. Second, large resource overheads are involved for Lockstepping since most such techniques are based on the core level redundancy. In contrast, RMT saves huge redundant resources since it can be implemented in a single chip using multiple hardware threads, but it comes with increased design and verification efforts on the controlling between the copies. RMT is more robust than Lockstepping against CMF errors due to the high design diversity from modules with different realizations. In the following, prevalent implementations using both techniques are presented.

- ***Stratus ftServer* [177]** targets mission critical applications which have very low SDC and DUE rates. The lockstepped system adopts its sphere of replication including off-the-shelf cores, main memories, I/O subsystem and fault detection modules. It supports the configurations of both DMR and TMR modes.
- ***Hewlett-Packard NonStop Himalaya* [207]** is implemented using Lockstepped MIPS microprocessors. The sphere of replication includes the MIPS cores, secondary caches and ASIC interfaces for fault detection by signal comparison. The main memory and I/O subsystem are out of such sphere. The Hewlett-Packard server takes advantage of the Lockstepping by process pairs in the kernel of its operating system.
- ***IBM Z-series* [179]** defines the replication sphere to be the processor pipeline, including instruction fetch/decode and execution units. The fault detection unit is moved out of the sphere to reduce the critical path. The authors in [179] estimate an area overhead of 35% from this Lock-stepped implementation.
- ***AR-SMT* [154]** is a single-core implementation of RMT technique incorporating two threads: the active A-thread and redundant R-thread. The committed data values from A-thread are kept in a delay buffer to be checked by the instruction stream from the R-thread. The sphere of replication includes the register file and the main memory, which achieves good memory fault coverage at the cost of two physical memories.

- **DIVA [12]** achieves RMT with a simple checker processor to detect errors in a superscalar core. The checker core incurs a relatively small area overhead, which is 6% for an Alpha processor [203]. The independent checker core enables DIVA to detect design failures, thus named as *dynamic implementation verification architecture*. One drawback in the design of DIVA is that the checker core is always assumed to be correct. In case of mismatch the result of the checker core is adopted. Transient faults in the checker core itself is not addressed. Besides, DIVA cannot detect error from the decode stage.
- **Argus [124]** applies similar technique as DIVA by the extension of a simple RISC core. Instead of replicating all instructions, it only verifies control flow, data flow, computation and memory interfacing instructions. The experiment shows that only 17% area overhead of the RISC processor overhead is imposed to achieve the fault coverage of 98%.
- **URISC [148]** realizes the RMT protection by a ultra-reduced instruction-set co-processor, which has only one turing complete instruction *subleq* from the MIPS ISA. Different instructions in the main core is protected by different sequences of *subleq* instructions. URISC achieves 30% area overhead than its original MIPS core. Due to URISC's difference in decoding instruction sequences of the main core, it achieves good fault coverage for errors in the decoder.

Most previous works achieve hardware redundancy by core-level duplication while some exploits multi-thread implementation within single core. However, techniques in AR-SMT are still expensive for the embedded processors since it does not support multithreading mechanism. A low-cost implementation of SRT for embedded RISC and VLIW processors is presented using the concept of opportunistic redundancy by the existing resources. The details are presented in Section 6.1 [211].

3.3.1.2 Information Redundancy

Information redundancy or coding technique, has been widely used for protection of memory-like structures, which is projected to exceed 70% of the die area by 2017 and cause most reliability related problems [168]. Parity and Single Error Correction Double Error Detection (SECDED) are two fundamental techniques in the realm of Error Correction Code (ECC) due to their simple implementation. Parity bit is one single bit for counting whether the encoded data word contains even or odd number of ones, which is used only for error detection. SECDED is encoded and decoded by using generation and checker matrix in linear time. In case of an detected error bit, the syndrome is calculated to detect the error location in order to correct it by flipping its value. A typical implementation of SECDEC is Hamming code. For 32 bits data, 6 bits of hamming codes are necessary. For details in coding theory and its application the readers are referred to the book by Peterson and Weldon [141].

ECC has been investigated heavily for the mainstream processors. IBM introduces the concept of Chipkill-correct [46], which interleaves the ECC coding such that two

consecutive data bits are encoded in two different code words. The approach is able to protect the memory data facing complete damage of single memory bank. AMD further develops such technique to reduce the required memory rank while achieves the same level of protection [22]. In [189] novel implementations of Chipkill-level reliability are proposed for efficient future memory devices. Other than the traditional SECDED codes, other coding techniques such as BCH codes are proposed to protect the memory system from more bit errors [205]. An efficient implementation of BCH is presented in [111].

In Section 6.2 [200] an alternative technique for multi-bit correction is presented by extending the standard SECDEC for fine-grained data segments according to their criticality. Different schemes of asymmetric protection are illustrated and demonstrated on embedded RISC and VLIW processors.

3.3.2 Approximate Computing

Recent research shows the trend towards exploring the energy-QoS trade-off based on the observation that huge amount of energy has been spent on guaranteeing exact correctness. However, exact correctness is not always required due to several characteristics of the applications. For example, computational intensive applications such as recognition, data mining and synthesis (RMS) use probabilistic algorithms, which use probability values or probability densities to compute or represent information. The effects of inaccuracies can be reduced over many iterations or by using a large number of samples [72]. Furthermore, applications such as video and audio processing exhibit the feature of cognitive resilience due to the limitation of human perception. In [33] an framework to characterize application resilience is presented. Consequently, approximate computing or inexact computing techniques, which exploit application-level characteristics for energy saving, become prevalent in research. In this section a survey on the relevant techniques from different design abstractions are illustrated.

3.3.2.1 Circuits-level Techniques

- *Kahng@2012* [95] presents an accuracy-configurable approximate adder (ACA) where the accuracy of results is configurable during runtime. Due to its reconfigurability, the ACA adder can operate in both approximate mode and accurate mode. The result shows that the ACA adder achieves 30% power reduction compared to the conventional adder with the relaxed accuracy requirement.
- *IMPACT* [70] proposes various approximate full adders with reduced complexity at the transistor level, and utilize them to design approximate multi-bit adders. The reduction in switch capacitance also gives in shorter critical path which provides additional chances for frequency scaling. Results which adopt proposed adder for image and video compression algorithms indicate the power savings of 60% and area savings of 37% with a small loss in output quality.

- *Miao@2012 [126]* introduces a novel approximate adder structure using an aligned, fixed internal-carry structure for higher significant bits. It also proposes conditional bounding as an optimization technique for synthesis of lower significant bits. The proposed adder achieves up to 60 % energy saving compared to the conventional timing-starved adder.
- *Kulkarni@2011 [102]* presents a 2x2 under-designed multiplier block and shows its usage for building arbitrarily large power efficient inaccurate multipliers. The architecture is tunable while the errors can be corrected at the cost of power. The approximate multipliers achieve an average power saving up to 45.4% over conventional multiplier with an average error up to 3.32%.
- *Razor [57]* demonstrates a novel pipeline structure which enables dynamic voltage scaling by monitoring the error rate during circuit operation. The goal is to eliminate the need for voltage margins based on the instruction and data dependence of circuit delay. A Razor flip-flop is proposed to double-sample pipeline stage values by a fast clock and a delayed clock. The value in the fast flip-flop is compared with the one from the delayed flip-flop to check metastability error. A pipeline mispeculation recovery mechanism recovers correct program state once upon a timing error is detected.
- *Constantin@2015 [36]* proposes an approximate processor pipeline structure with dynamically adjustable clock, which is set according to dynamic timing analysis of different instructions and operands. The approach enables frequency overscaling without timing errors. Results show that 38% of speed increment or 24% power reduction is achieved.

3.3.2.2 Architectural Techniques

- *ERSA [72]* presents Error Resilient System Architecture targeting RMS applications. The proposed heterogeneous multi-core system has several features. First, cores are designed with asymmetric reliability which contain super reliable core (SRC) and relaxed reliable core (RRC). ERSA uses expensive SRC for executing non-error-resilient portion of applications, while cheap RRC for portions of application which contain approximate features. Second, low cost boundary checkers are adopted for memory access and timeout errors. Third, software techniques are introduced to modify the applications with minimal intrusiveness. The prototype of ERSA shows that 90% of output accuracy is achieved under a very high soft error rate.
- *Chippa@2010 [34]* implements accuracy scaling mechanisms from high-level abstractions using control knob in the architecture. Three types of accuracy control are applied which are voltage over-scaling at circuit level, dynamic precision control at architectural level and significance-driven algorithmic truncation at application level. Greater energy saving are gained by synergistically co-optimizing across different abstractions.

- **Chippa@2011** [32] proposes a general framework by dynamically regulating scaling mechanisms according to the quality requirement. Low-overhead sensors are used to estimate output quality, while a feedback control mechanism tries to maintain output quality within a specified range using the control knobs similar as the ones in [34].
- **Georgios@2012** [97] tunes the degree of voltage over-scaling for individual block of the DSP system based on user specifications and severity of process variations/channel noise. Minimum system power is ensured while adequate quality is provided. Cross layer approaches of unequal error protection are applied for tuning both logic and memory modules. 69% improvement in power consumption is achieved for reasonable image quality.
- **Banerjee@2007** [17] designs an novel DCT architecture which tolerates strong process variations. The key idea is to limit the erroneous effect of process variation under voltage over-scaling to the long paths which contribute less to the PNSR improvement, yet offering a large improvement to power dissipation with small PSNR degradation. The results show a 62.8% of power saving, which is achieved by a gradual quality degradation under large process variation and low supply voltage.

3.3.2.3 Synthesis Techniques

- **SALSA** [194] exploits quality trade-off during logic synthesis of generic circuits. The approach encodes quality constraints as Q-functions which takes advantage of the *Approximation Don't Cares (ADC)* from the primary outputs. ADC based analysis enables the circuits simplification using the traditional Don't Care based logic optimization techniques. Significant area and power savings are achieved through the approach.
- **ASLAN** [149] is the first approach to synthesize approximate sequential circuits. ASLAN formulates the quality based synthesis as a sequential model checking problem by identifying liveness and safety properties in the circuits which guarantee the correctness of the approximate circuits. It also maximizes energy saving for a given output quality using SALSA-based technique for synthesizing the combinational blocks.
- **MACACO** [195] proposes a systematic methodology to analyse the behaviors of approximate circuits using metrics such as worst case error, average error, error probability and error distribution. The approach is taken by conventional Boolean analysis techniques such as SAT solver and BDD for an untimed circuit representing behavior of the approximate circuit. SAT solver predicts the worst case error while BDD gives the error distribution.
- **GALS** [125] formulates that the approximate logic synthesis problem unconstrained by the frequency of errors is isomorphic to the Boolean relations minimization

problem, which is then solved by algorithms of Boolean relations for the error magnitude-only constrained approximate synthesis problem. Furthermore, a heuristic algorithm is proposed to iteratively refine the magnitude-constrained solutions with the purpose to finally make the solution satisfying the error frequency constraint. Experiments show that 60% of literal reduction is achieved for tight error magnitude and frequency constraints.

- **SASIMI [193]** provides another optimization technique during logic synthesis by identifying signal pairs in the circuit which exhibit the same value with high probability and substituting one for the other correspondingly. The fanout circuits of the logic being removed is consequently downsized due to extra timing slack. The approach ensures the input quality constraints and iteratively performs substitution automatically.
- **Probabilistic Pruning [113]** first introduces a ranking function to rank the significance and activity of nodes in the circuits. After that, the logic pruning is performed by iteratively removing nodes with least ranking until target error bound is achieved. Results on a 64-bit adder show up to 7.5x gain in the Energy-Delay-Area product with up to 10% of error percentage compared with conventional design.
- **Probabilistic Logic Minimization [114]** is another ranking based optimization technique during logic synthesis by intentionally bit-flipping elements in the logic look-up-table to achieve potential literal and operator minimization. The bits for flipping are selected based on the ranking of lowest input combination probabilities from the application-level characteristics. Results on a 16-bit ripple carry adder and array multiplier show up to 9.5x gain in the Energy-Delay-Area product with up to 1% of error percentage compared with conventional design.

3.3.2.4 Programming and Compilation Techniques

- **EnerJ [164]** proposes type qualifier for variables involved in approximate computing. Such variables are automatically mapped to low-power storage, operations and energy-efficient algorithms. The system isolates the precise variables from the approximate ones, where the users can explicitly control the casting from approximate type to precise type of the variables. Using EnerJ in Java programs leads up to 50% of energy saving with little accuracy cost.
- **Truffle [58]** proposes another microarchitecture design supporting instruction extensions for quality aware programming. The quality selection is implemented by dual-voltage operations. The architecture contains approximate execution units, registers, caches and main memories, which are exposed to selection on instruction-level granularity. Energy saving of up to 43% is demonstrated for several benchmarks.

- **QUORA [192]** presents an energy efficient, quality programmable vector processor using hardware based precision scaling and error compensation mechanisms. QUORA contains a separate set of instructions implementing quality aware instructions. The architecture contains approximate processing elements and accuracy processing elements for different computing accuracy requirements. Simulation result shows up to 1.7x energy saving with less than 0.5% loss in application quality.
- **GREEN [16]** introduces a systematic approximate programming approach with two phases of operation. The calibration phase builds a model of the QoS loss produced by the approximation, which is applied in the operational phase to make approximation decisions based on the QoS constraints. An adaptation function is included in the operational phase which monitors the runtime behavior and updates the approximation decisions to guarantee statistical QoS. The proposed approximation techniques and language extensions are integrated into the Phoenix compilation framework and demonstrated for the energy saving on graphics, machine learning, signal processing and web searching applications.
- **Shafique@2013 [170]** takes advantage of program-level error masking and propagation properties to perform reliability-driven instruction prioritization and selective protection during compilation. Statistical instruction-level error masking models are developed for estimating error propagation probabilities. Significant reliability improvement is achieved compared with state-of-the-art reliability techniques during program compilation.

In Section 6.3, the state-of-the-art programming and architecture techniques are enhanced by an alternative method for mitigating memory failures and presenting the necessary software and hardware features for its realization within a RISC processor. By focusing on memory faults, rather than correcting every single error, the proposed method exploits the statistical characteristics of any target application and replaces any erroneous data with the best available estimate of that data.

3.4 System-level Fault Tolerant Techniques

The advent of multi-processor system-on-chip provides new design opportunities for applications with high performance and low power requirements. At the same time, system-level fault tolerant techniques address the reliability issues for MPSoC in parallel with architectural and circuit-level techniques. In this section, two classes of system level fault tolerant techniques are briefly discussed which are reliability-aware task mapping and reliable network design.

3.4.1 Reliability-aware Task Mapping

Continuous performance scaling tends to decompose applications on MPSoC into small tasks, which can be executed in parallel on the multiple cores and communicate

with each other. The problem of task mapping involves an optimal task deposition and scheduling mechanism in favour of performance/power/reliability constraint. Generally, task mapping approaches can be viewed from different perspectives. First, according to the time instance when mapping takes place they can be categorized as design-time mapping which is used for static workloads and run-time mapping which includes dynamic workloads. Second, based on the types of architecture components they can be sorted as homogeneous and heterogeneous mapping. Besides, for dynamic workloads task managers are required, where the control mechanisms of task manager can be either centralized or distributed. Furthermore, intensive computation efforts are required to compute the decision of run-time mapping especially for many-core processor systems. Such computation would place strong performance overheads for the task manager if it is performed online. An alternative solution is to calculate the mapping results by design time analysis and to book-keep such results in the storage of task manager, so that the mapping scenarios are read directly when decisions need to be made. A survey from [173] is referred to for details on the various mapping strategies.

Within the large amount of research for task mapping methodologies, reliability-aware mapping becomes a hot research topic in nanoscale computing recently. With regard to the techniques *improving device lifetime*, [44] discusses the proper approaches to address the lifetime optimization in terms of mean time to failure. Coskun et al. [39] discusses temperature-aware mapping that leads to increased lifetime. A wear-based heuristic is proposed in [78] to improve the system lifetime. On the other hand, several works target the field of reliable *mapping for transient faults* where the cores are temporarily corrupted. In [106] the author proposes reliable remapping technique aiming at determining task migrations with the minimum cost while minimize the throughput degradation. In [166] a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems is presented. The task manager moves the tasks on the failure core onto available cores allocated during design time. [47] demonstrates several fault tolerant mapping algorithms by using Integer Linear Programming (ILP) under faulty core constraints. The algorithms tend to minimize the communication traffic and the total execution time caused by the permanent failures. ERSA architecture [72] introduces the asymmetric mapping technique which allocates critical task portion to highly reliable core while the rest task portions to less reliable cores manually from application designers. Enlightened from ERSA, in Section 7.1 [198] a heuristic mapping algorithm considering various task criticality and core reliability levels is proposed and demonstrated in a system-level reliability exploration framework.

3.4.2 Fault-tolerant Network Design

In contrast to task mapping techniques where the network topology for the MPSoC is predefined, fault-tolerance in network design involves the reliability evaluation of network topology according to its graph structure from theoretical perspectives. Different reliability targets are presented in the literatures such as ensuring connectivity,

least routing overheads, distance guarantee and ensuring graph isomorphism in presence of failure nodes or edges. Selective works in this fields are presented in the following.

- **Group graphs [5]** presents the reliability analysis of a set of graphs named Group Graphs, which are constructed based on symbol permutations. The work demonstrates that most group graphs exhibit optimal fault tolerance in the sense that each node in the graph is still able to connect to all other nodes when $d - 1$ nodes are removed from its neighbouring nodes, where d is the amount of its neighbouring nodes. Besides, the symmetric property of group graphs automatically alleviates many interconnection problems such as congestion and message-routing.
- **Generalized de Bruijn Graph [80]** discusses its fault-tolerant properties in terms of the latency and energy consumption cost by a link failure. The work demonstrates that such latency of Generalized de Bruijn Graph is much less compared to Mesh and Torus style topologies. The reason lies in the logarithmic relationship between the diameter of the graph and the number of nodes in the graph. Besides, the graph's Hamiltonian nature also contributes to its fault tolerance ability.
- **Graphs with Distance Guarantees [75]** addresses the graph reliability by finding the subgraph structure named as k -spanners from an arbitrary graph. K -spanners gives an upper bound to the graph distance between any nodes in the graph, which is applied to construct fault tolerant graphs that guarantee constant delays even if a multiple number of edges fail.
- **Node fault tolerance [77]** approaches the problem of constructing reliable network topology by designing a supergraph which is isomorphic to the task graph when any of its nodes and connecting edges is removed. To find such supergraph with smallest amount of edges is the problem of optimal node fault tolerance (NFT). Harary proposes several techniques to build optimal NFT graph for selective graphs, including path, circle and a set of tree structures. Such techniques are generic in the sense that the supergraph can tolerate any number of failing nodes.
- **Edge fault tolerance [76]** addresses a similar problem as in [77] to find optimal Edge fault tolerance (EFT) supergraph from a given task graph, which contains smallest amount of edges out of all EFT supergraphs. Families of optimal EFT graphs are presented for n -node path or cycle with k edge tolerance.

In Section 7.2, the methodology in [77] is extended to construct optimal NFT for arbitrary graphs by decomposing them into small graphs which are individually handled using the original NFT theory. An exhaustive search based heuristic algorithm is designed to verify the correctness of the proposed approach and reduces the searching space for optimal NFT graph significantly.

Chapter 4

High-level Fault Injection and Simulation

In this chapter the fault injection technique for generic architecture models is presented, which provides an experimental setup for reliability estimation and exploration. First the architectural fault injector is illustrated in Section 4.1. After that the system-level fault injector is presented in Section 4.2. Finally a case study which connects fault injection with power and thermal simulation is described in Section 4.3.

4.1 Architectural Fault Injection

Although faults can be simulated accurately only at the circuit level of abstraction, there have been many proposals to inject faults at high levels of abstraction. Fault injection techniques can be divided into three categories, i) physical fault injection, ii) software fault injection and iii) simulated fault injection. Physical fault injection introduces the fault in the hardware by emulating a faulty environment [88] or by doing ground testing [145]. Software-based fault injection techniques alter the processor state (memory, register) to simulate a fault. Detailed account of such methods are presented at [15]. Simulated fault injection approaches simulate the microarchitecture, usually at Register Transfer Level. Several fault injection tools in this category are proposed [42] [91] [18]. While pure software fault injectors suffer from restricted view of the microarchitecture, RTL-based simulated fault injectors are usually slow and in some cases need to be recompiled. In order to strike a balance, fault injection can be performed during instruction-set simulation. Processor instruction set simulators offer different degrees of accuracy and speed trade-off. This is leveraged by creating a fault injection set up based on cycle-accurate instruction-set simulator in [120] and [118].

The lack of generic tools for fault injection at architectural level enforces the designer to include at least the RTL abstraction in the design space exploration flow or stick to a specific processor design environment, which impedes early design space exploration. Error detection and correction mechanisms at architectural level currently do not fully exploit the processor knowledge. For example, the reliability requirement at the processor decoder, processor datapath and the processor storage can be different. Software-based fault detection and protection mechanisms also need to be aware of the available hardware methods much like the traditional compiler-microarchitecture design problem. These issues can only be addressed by considering reliability at early design space exploration phase. For high-level processor design,

a large body of research work is presented. In particular, with the entry of Architecture Description Languages (ADLs), processor design has not only become easier but it also offers high-level design space exploration capabilities. This has shown to improve design quality and productivity tremendously, thereby gaining widespread commercial acceptance [182] [184].

Contribution This work attempts to keep the system generic so that arbitrary processor architecture can be modelled and simulated with proposed fault injection framework. More importantly, the fault injection is designed in a high-level processor design environment based on ADL LISA (Language for Instruction Set Architecture) [2], which is used for early design space exploration. One can add a fault protection mechanism in hardware or in software and quickly evaluate the corresponding area or timing overhead as well as test the efficacy of the protection. Although experimented using ADL LISA, the proposed techniques are general enough to be applicable to any high-level processor design environment.

4.1.1 Methodologies

The proposed fault injection technique is presented in this section. Brief overview of LISA language is first introduced, which is followed by the description of fault models covered in this work. The fault injection approach on LISA processor model is then explained. After that the method of error evaluation is presented.

4.1.1.1 Brief Overview of LISA

In LISA, the OPERATION is the key element to describe the instruction set, timing and behavior of the processor. One instruction can be hierarchically split into several OPERATIONS, each of which describes one part of the instruction. The OPERATION contains several sections such as CODING, SYNTAX and BEHAVIOR which represent the encoding, syntax and behavior of the instruction respectively. The BEHAVIOR section encloses plain C codes so that arbitrary functionalities can be specified. The C code allows user-defined data types such as bit types.

The declarations for processor resources such as pipeline stages, registers, memories and ports are located in the global RESOURCE section of LISA and they can be accessed from the LISA OPERATION. To describe the microarchitecture, structural information can be added by assigning the operations to different pipeline stages outside the RESOURCE section. This essentially covers the scheduling. The microarchitecture and the Instruction Set Architecture (ISA) are completely described by RESOURCES and OPERATIONS. For further information on LISA language please refer to [2] [182].

4.1.1.2 Fault Models

To represent physical faults which occur in real integrated circuits, several fault types are currently implemented which are presented in Table 4.1. In the table, t is the

evaluation time, which is the time between fault starting time t_s and fault ending time t_e . $V(t)$ is the original signal value before fault injection and $F(t)$ is the signal value after fault injection. While Table 4.1 explains the implemented transient fault types, the permanent faults can be easily realized by setting t_e to the end of the simulation time. Three kinds of bit-flip faults are implemented. Instantaneous bit-flip assigns the resource with faulty value at specific time instance and then instantaneously releases the control of that resource to the simulator. SEU can be modelled using this fault. Bit-flip with duration keeps the faulty value for several clock cycles before releases the control to the simulator, which resembles intentional fault injections such as in fault attacks. For the toggling bit-flip fault type, the faulty value toggles with the original value at each clock cycle during the fault injection period. This is different from the other bit-flip faults where the faulty value is the inverted value of the signal at the fault starting time. Other fault models can be easily extended through the interface of fault modules.

Table 4.1: Currently implemented fault types

Fault Type	Expression for fault value ($t_s < t < t_e$)
Stuck-at 0	$F(t) = 0$
Stuck-at 1	$F(t) = 1$
Instantaneous bit-flip	$F(t_s) = \text{Not}(V(t_s))$
Bit-flip with duration	$F(t) = \text{Not}(V(t_s))$
Toggling bit-flip	$F(t) = \text{Not}(V(t))$
Indetermination	$F(t) = X$
High Impedance	$F(t) = Z$

4.1.1.3 Fault Injection Method

In this work, simulated fault injection technique on LISA simulator is developed. To overcome the inefficiencies of the SC method, a hybrid approach which combines both SC and CM methods is adopted. In the instruction set simulator generated by the LISA compiler the reachable signals from the simulator are limited to the processor resources defined in LISA resource sections. While SC technique can be applied to change values of such resources through simulator interfaces, the local variables which reside in the LISA operations are still unreachable. To solve this problem, additional LISA signals are declared which help the simulator to reach the local variables. Such signals work as disturbance to the local variables inside the LISA operations. In this approach the controllability of SC method of LISA simulator is extended. Depending on the usability of fault simulation, user can choose whether SC technique is used standalone or the hybrid approach is needed.

4.1.1.4 Error Evaluation Method

The *Error Manifestation Rate* (EMR) [42] is used as a metric for the evaluation of fault simulation. Suppose one fault is injected randomly both in time and location into the device model or a component of such device in each experiment, *EMR* is defined as the percentage of experiments which detects error on the memory interfaces to which the faults propagate. Mathematically, given the number of total experiments as N_i ,

$$EMR = N_e / N_i \times 100\% \quad (4.1)$$

where N_e is the number of experiments with error detected. Normally, the *EMR* increases with the duration and number of injected faults. For the same fault duration and number among different components, larger *EMR* value reflects higher error probability, which means less reliability of the faulty component. The interface values are traced and compared with values from fault free *golden* simulation for error detection. Besides, user can define their tracing signals for further analysis of the target architecture under faulty environment.

Note that *EMR* does not directly determine the percentage of user visible error which is highly dependent on the application characteristics. However, it reflects the architecture level error resilience ability within the processor. In case that no timing perturbation is caused by pipeline stall and rollback, the cycle-based comparison of interface signal values with the ones from golden simulation indicates the error manifestation.

4.1.2 LISA-based Fault Injection

Based on the technique in previous section, LISA-based fault simulation framework is discussed here in detail. Figure 4.1 shows an overview of the framework. It consists of the following three phases.

4.1.2.1 The setup phase

In this phase, the processor model and fault configuration for the simulation and error evaluation are generated. By using LISA compiler, simulation models for the LISA processor descriptions are generated. From the processor model, all the declared resources such as registers, memories and signals are exposed to fault injection. In case of hybrid fault injection, the declarations of disturbance signals and modifications inside the behavior sections of LISA operations are required. Figure 4.2 shows such code-modification method by an example which performs *add* operation in the EX pipeline stage of a RISC processor. Specifically, all the read identifiers in each behavioral statement can be masked by simple logic operations with the disturbance signals, whose values are assigned later according to user configurations. The masking operations can be easily modified through pragmas so that different fault types can be realized. In Figure 4.2 this is the case how signals *alu_in1* and *alu_in2* are masked to realize the bit-flip type of fault.

Figure 4.1: LISA-based fault injection and evaluation flow

Besides LISA behavior sections, the activation signals used to implement scheduling for LISA operations can also be disturbed. A single bit disturbance signal is tested to determine whether the target operation would be activated. This is also shown in Figure 4.2.

To generate the fault configuration file a graphical user interface is developed, which converts user defined fault properties into a configuration file in XML format. Figure 4.3 takes a snapshot of the graphical interface. A list of implemented fault properties is presented in Table 4.2. The fault injection tool also supports the capability to annotate the Bit Error Rate (BER) for each bit of the resources. The BER can be characterized from low-level simulation or taped-out circuits. For instance, under different run-time frequencies, the probability of delay faults on individual logic path differs. Fault injection with resource specific fault probabilities facilitates architecture design with real physical parameters.

The unit for injection time and durations are the number of clock cycles due to the usage of cycle-accurate simulation. Several fault properties are specified in a range

<pre> OPERATION add IN pipe.EX { BEHAVIOR { alu_out = alu_in1 + alu_in2; } ACTIVATION { write_back } } </pre>	-----	Without Fault Injection
<pre> #define FT ^ // Options: s-t-1: , s-t-0: &~, bitflip: ^ RESOURCE { uint32 fault_add_1; // Disturbance signals declarations uint32 fault_add_2; bit[1] ft_activation_add_1; } OPERATION add IN pipe.EX { BEHAVIOR { // Disturb the original signals alu_out = alu_in1 FT fault_add_1 + alu_in2 FT fault_add_2; } ACTIVATION if (!ft_activation_add_1) {write_back} } </pre>	-----	With Fault Injection

Figure 4.2: Logic fault injection through disturbance signals

between the minimal value and the maximal value. The purpose of doing this is to facilitate statistical fault injection where properties of the faults are determined statistically to model faults with distribution. An additional option associated with each specified range is created to select the distribution functions. Currently uniform and normal distributions are supported while extensions are easily achievable through user defined probability density function. The location of injected fault can be fine-grained into one specific bit. Such implementation eases the experiments such as bit-level fault attack. If no injection location is specified, the tool will randomly select the location from all the processor resources. The fault types can be chosen from the list specified in Table 4.1 which are defined in the fault library. Further extensions to the fault types can also be made by adding new faults in the fault library. The distributions for several fault properties can be specified to facilitate probabilistic fault injection.

Figure 4.3: Graphical user interface for fault configuration and evaluation

Besides configurations of fault injection, Table 4.2 also shows the configurations for the error evaluation, where simulation time and total number of experiments provided as user inputs. The resource list for *Value Change Dump*(VCD) tracing provides a convenient interface to define error condition based on user needs. By default, the enable, address and data signals from all instruction and data memories are traced to detect potential errors. List 4.1 presents an example of fault configuration file in XML format.

To facilitate fault injection on generic LISA models, user can specify the target architecture model and the application file. An architecture specific fault simulator will be generated based on the configurations.

4.1.2.2 The simulation phase

In this phase the fault injection is performed during behavioral simulation. The Synopsys Processor Designer provides convenient programming interface [182] for the user to simulate and interact with the processor models. In this work, several methods such as model initialization, cycle based execution, resource value tracing, get and set resources values are used to inject the faults based on user configurations. The simulation phase consists of four steps.

First, the fault configuration file is parsed, where data structures are created for both fault simulation and evaluation. Each defined fault is allocated a data structure to store its information such as inject time, duration and location. All such structures are appended onto a wait fault list while another active fault list is initialized to be

Table 4.2: Fault properties in configuration file

Purpose	Contents
General	Target architecture model Running application file
Fault injection	Injection time range (unit clock cycles) Fault duration range (unit clock cycles) Fault types Fault locations Bits range of resource Array range of resource Number of injected faults Probability distribution of fault properties
Fault Evaluation	Simulation time (unit clock cycles) Total number of experiments List of tracing resources

```

<ConfigFaultSim>
  <ConfigFaultInjection>
    <ConfigPlacesAndFaults InjectionMode="Probability">
      <ConfigFaults>
        <Mode>EMR Trend with Faults Duration</Mode>
        <InjectCycleLower>50000</InjectCycleLower>
        <InjectCycleUpper>50100</InjectCycleUpper>
        <DensityFunOfInjectTime>Uniform</DensityFunOfInjectTime>
        <FaultsNumberLower>10</FaultsNumberLower>
        <FaultsNumberUpper>10</FaultsNumberUpper>
        <DensityFunOfFaultNo>Uniform</DensityFunOfFaultNo>
        <FaultsDurationLower>1</FaultsDurationLower>
        <FaultsDurationUpper>1</FaultsDurationUpper>
        <DensityFunOfFaultDuration>Uniform</DensityFunOfFaultDuration>
        <ConfigInjectPlaceName>R</ConfigInjectPlaceName>
        <ConfigInjectPlaceIndexLower>4</ConfigInjectPlaceIndexLower>
        <ConfigInjectPlaceIndexUpper>4</ConfigInjectPlaceIndexUpper>
        <ConfigFaultType>Bit_Flip_Transient</ConfigFaultType>
      </ConfigFaults>
    </ConfigPlacesAndFaults>
  </ConfigFaultInjection>

  <ConfigFaultAnalysis>
    <DumpVcdFileName>lisa .dump</DumpVcdFileName>
    <SimulationCycles>1200</SimulationCycles>
    <TracedResources>
      <ResourceName>R</ResourceName>
    </TracedResources>
  </ConfigFaultAnalysis>
</ConfigFaultSim>

```

Listing 4.1: Example of fault configuration File in XML

empty. The wait fault list is sorted in ascending order of injection time. Second, the tracing resources specified in the configuration file are added to the tracing list of Processor Designer. When the simulation starts, a scheduler is triggered to inject the faults based on the injection time of the elements in the wait fault list. When the simulation reaches the cycle instance when a fault needs to be injected, the data structure of such fault is moved from the wait list to the active list. For each clock cycle when a fault is injected, the remaining injection time of such fault is reduced by one. The fault is deleted from the active fault list when its remained injection time reaches zero. When both the wait list and the active list become empty, the injection of all specified faults are completed. The executed program finally runs until the simulation time is over. The golden simulation is easily realized by emptying the wait fault list at the beginning of the simulation.

4.1.2.3 The evaluation phase

In this phase the tracing results of interface signals are compared and analysed. To get the value of *EMR* for a number of experiments, the toggling information on the ports of processor interface is traced and stored into VCD file during each experiment. Also, the VCD file for the golden simulation without fault injection is traced to compare with the faulty one to detect potential errors. After each experiment, the result of such comparison is recorded and the total number of experiments with error detection is updated. When all experiments are completed, the *EMR* is calculated according to Equation 4.1.

The evaluation results contain the *EMR*, a logfile recording the fault injection events and comparison of VCD files between golden and faulty simulation for each experiment.

4.1.3 Timing Fault Injection

In most high-level simulation frameworks, clock cycle is adopted as the notion of time, which is the smallest time unit which maintains a stable processor state. Such framework lacks the ability for physical timing simulation, which is usually generated for a post-layout circuit netlist. To integrate low-level timing as a constraint for delay-based fault injection, LISA-based processor simulator is extended with timing annotation of the logic paths, which are extracted from the timing analysis files. Such annotated path timing will be compared with runtime clock period, so that delay faults can be injected. This subsection illustrates the simulator extension for timing fault injection.

4.1.3.1 Simulation kernel

To facilitate timing fault injection, the simulation kernel is extended by the modules in Figure 4.4.

- Initial timing for logic paths. It shows the bitwise logic delay from initial flip-flop to the end flip-flop of a logic path, which are analysed during logic synthesis

Figure 4.4: Simulator extension for injection of delay faults

or placement and routing using Static Timing Analysis (STA) [79]. Such bit-wise delay information is back-annotated as extra information for the hardware resources in instruction-set simulator.

- Timing variation model, which updates the runtime delay based on initial delay and user provided timing variation function. For instance, temperature aware timing variation function is provided when the path timing is changed through temperature and time. In Dynamic Timing Analysis (DTA), this could be the timing look-up-table which stores information on instruction dependent path timing variation.
- Running frequency, which is defined by the user to compare with the runtime delay, so that a fault could be injected. A runtime adjustable frequency can be applied to model Dynamic Frequency Scaling (DFS).

For each simulation clock cycle, the simulator first updates the clock cycle time using the initial delay and timing variation function for all annotated paths. In the next, the simulator checks timing violation for all annotated logic paths. In case there is a timing mismatch, the simulator overwrites the current value in the target resource by a random value which is either zero or one to model metastability or the value from previous clock cycle to model a delayed logic latching. Otherwise, the simulator stores the current resource values which may be used as fault injection value for the following clock cycles.

4.1.4 Experimental Results

Two case studies are presented in this section. To demonstrate the effectiveness of the fault injection framework it is first benchmarked with a state-of-the-art HDL-based fault injection. Using an example of RISC processor customized for cryptographic application, the second case study shows how fault prevention techniques can be quickly explored with the help of the proposed framework.

4.1.4.1 Benchmarking with HDL-based Fault Injection

In [42] an HDL-based fault injection framework is presented, which is based on Verilog Programming Interface. The fault injection is realized by scheduling events in the event queue of Verilog simulator. By providing gate-level netlist and standard cell library the framework in [42] can perform gate-level fault simulation. For cycle-accurate simulation on ADL abstraction such gate-level refinement is missing. Consequently, the proposed technique is compared with the RTL fault simulation provided by their work.

Experiments are carried out on a RISC and a processor with 5 pipeline stages, which are parts of IPs distributed with Synopsys Processor Designer. The testing application is *Sieve of Eratosthenes* which is used to generate prime number in a specified range of numbers. After compilation of the LISA descriptions, LISA-based fault injection can be applied directly on the generated processor simulator. To use the HDL-based fault injection framework, the Verilog codes are generated from the LISA descriptions by the processor generator. In the following the comparison with regard to the accuracy and speed is discussed.

Accuracy To demonstrate the fault injection accuracy for the RISC processor a group of fault configurations are specified. Bit-flip type of faults are injected into 6 Verilog modules randomly in time and location. The number of single bit bit-flip faults in each experiment is changing from 1 to 6 while the fault duration is kept at 1 clock cycle. The simulation time is set to 1200 clock cycles. Each measurement point averages 3,000 fault injection experiments for an adequate statistical estimation.

Figure 4.6 shows the EMR trends with number of faults. It can be observed that LISA-based fault simulation framework provides similar results as RTL fault simulation. Differences in absolute values for the same fault number are due to two factors.

First, since both injection frameworks generate their faults information independently, two random sets of experiments at LISA and RTL differ from each other with regard to injection time and location. Second, in most cases LISA fault injection gives slightly higher value than RTL injection. This happens because signals which have minor effects in fault simulation are missing in LISA description, but are still available for the RTL injection, which reduces average EMR values.

Figure 4.5: Exemplary EMR with increasing duration of fault (RISC)

It is observed that the EMR increases faster with increasing number of faults than increasing duration. This happens because multiple bits of injected faults are uncorrelated with each other in both injected time and location. However, single bit fault with long duration has a constrained injected location which limits its increment of error probabilities compared with the multiple bits fault.

With regard to individual hardware modules, Fetch unit is most vulnerable among all six modules. This is because a fault inside it can potentially influence all instructions. Besides, WRITEBACK module gives lower EMR even though it is activated by most instructions. The reason is that during execution operands of instructions are mostly bypassed from pipeline registers other than obtained from the register files. Therefore an error in architectural register unnecessarily leads to an error on the memory interface due to architectural masking effect.

In order to verify the accuracy of fault simulation among different architectures, faults are also injected in VLIW processor for the same application. Figure 4.7 shows

Figure 4.6: Exemplary EMR with increasing number of fault (RISC)

its trend of EMR with fault duration. Besides similar EMR trends and accuracy, EMR values at the same evaluation points compared with RISC processor are reduced for all hardware modules except register file. The reason is that VLIW processor has four parallel datapaths and usually not all of them are busy during runtime. Faults randomly injected on the idle datapaths do not lead to an error. However, the number of registers in the register file are almost the same which affects the EMR less compared to the datapaths. The increased gap between EMRs for the register file is caused by the large number of access ports in the register file for VLIW architecture, which are not available in LISA resource section. The faults injected on the ports in RTL simulation are currently modelled as faults on the LISA resources. However, this may cause non-equivalent behaviors since a fault on the resource interface does not equal to a fault on the resource value itself. This leads to a simulation mismatch between LISA and RTL when the same resource value is accessed in the following cycles. Such gap is reduced for RISC processor when the number of ports are far less than that of VLIW.

Speed Speed is a major metric in the evaluation of any fault injection framework. Table 4.3 presents the time required to complete 3,000 experiments by both frameworks. Simulations are done on the same host machine while the simulation time for the application is 1200 clock cycles.

Figure 4.7: Exemplary EMR with increasing duration of fault (VLIW)

Table 4.3: Comparison of fault simulation speed

Frameworks	Time duration (3000 exps)
LISA-based Fault Simulation	935 sec
HDL-based Fault Simulation	9963 sec

It can be observed that LISA-based fault simulation achieves a 10x speed-up compared with the HDL-based one. This is due to the fact that simulations at higher abstraction levels are orders of magnitude faster [109].

4.1.4.2 Exploring Reliability using Fault Injection

LISA-based fault injection framework enables fast reliability exploration by high-level customization on hardware models. The automatic generation of instruction-set simulator also enables reliability exploration by software techniques. In this section, this is demonstrated by protecting a customized RISC processor from fault attack on Advance Encryption Standard (AES) [43] application.

The selected AES implementation is from Brian [27]. By the C compiler generated from Synopsys Processor Designer the source code is compiled into binary code which can be run on the RISC processor model. The bit-level fault model from [175] is used, where single bits of the temporary cipher result at the beginning of the final encryption round are flipped to get the cipher key. By using this method, the 128-bit AES key can be obtained by using less than 50 faulty cipher texts. The reliability is explored in following steps.

Vulnerable Resources Identification Methods have to be taken first to identify the vulnerable resources where the temporal cipher results between encryption round 9 and 10 are stored. The identification takes advantage of the fault-free simulation by which the executed assembly instructions and debugging information can be easily observed. It is found that such intermediate results are kept in register R[5], R[6], R[10] and R[11] during run-time.

Architecture Exploration First the fault protection based on software technique is quickly explored by directly modifying source code in C. The last 2 encryption rounds are executed twice and their results are compared to decide if an attack happened. Fault free simulation shows that the total execution time for encryption is increased by 1%. However, simulation also shows that the temporal cipher results of the additional 9th round are stored in the same registers as the original one. Thus the fault attack which lasts for a time period long enough to effect the results of both intermediate rounds is still undetectable.

To solve such problem, an alternative hardware based protection technique is applied, where the target registers which store temporal cipher text are simply duplicated. For this aim, 4 additional registers are declared in the resource section of processor model. To implement the protection mechanism, new features in logic operations are added. All logic operations writing to the target registers also write the same value to the duplicated registers. Besides, all operations reading from the target registers also read from the associated duplicated registers. Both read results are then compared with each other. The processor simply halts itself when mismatch is detected. Aforementioned architecture customizations are done in Processor Designer in only several minutes of time. After that, the updated processor simulator is generated automatically.

Fault Simulation Using the simulation framework, single bit-flips are injected onto the targeted registers to emulate real fault attacks. Result shows that the attack only on the target registers can not be successful but halts the processor, which demonstrates the efficiency of applied hardware reliability extensions.

HDL Generation and Logic Synthesis The HDL descriptions for the RISC processor is generated by the Processor Generator. To verify the results of protection HDL-based fault simulation [42] can then be applied. The architecture is synthesized using fara-

day 90nm technology library [59] by Synopsys Design Compiler [181]. Table 4.4 shows the synthesis result of both unprotected and protected architectures. Result shows a 3% increment in critical path length, 43% in area and 30% in power consumption compared with the unprotected architecture. The large amount of additional area is caused by the fault detection logic in the pipeline (contributed 24.5% of area increment) and protection registers (contributed 18.5%). The result reflects the trade-off between performance and security.

Table 4.4: Comparison of Synthesis Result

Architectures	Critical path (ns)	Area (KGates)	Power (mW)
RISC	1.41	23.0	11.5
RISC(hardware protection)	1.45	32.9	15.0

4.1.5 Summary

In this work, a LISA-based fault injection technique has been introduced. Both simulator-command and code-modification methods have been applied in this technique to increase fault simulation accuracy. The comparison with a state-of-the-art RTL level fault injection framework has shown the result of comparable accuracy and significantly less simulation time. By applying such framework, reliability exploration can be easily performed at the ADL abstraction, which facilitates fast prototyping of reliable architecture.

4.2 System-level Fault Injection

As the design complexity grows, Multi-Processor System-on-Chip (MPSoC) becomes the state-of-the-art architecture solution for high performance and low power applications. The design trend of complex, multi-core systems puts stronger focus on systematic high-level design flow than traditional Register Transfer Level (RTL)-based design flow. To this end, SystemC, which is a library of C++ functions supporting concurrent process simulation, has become the standard design approach for complex SoC modelling. On top of SystemC kernels, several platform simulation tools are proposed by Electronic System Level (ESL) tool vendors. It is necessary to integrate reliability exploration in system-level design. While there exists dedicated processor-specific fault-injection studies [35] [150], generic system-level fault injection is relatively less explored [119].

Contribution In this work, an efficient fault injection technique is presented for SoC components, such as processor IPs, bus and memory. The fault injection tools devel-

4.2.1 Fault injection for system modules

4.2.1.1 Processor fault injection

For processor-level fault injection, a commercial processor design flow [182] is adopted, which is based on the architecture description language LISA. Fault injection for LISA-based processors has been proposed in Section 4.1, where the LISA Application Programming Interface (API) is used for modifying the processor states during execution. To integrate LISA processor model into the platform, a SystemC wrap-

per is constructed which inherits all the methods including the API functions of the processor model. Taking advantage of LISA API interfaces, high-level processor models are subject to fault injection with sufficient accuracy. The system-level techniques apply the same methodology so that similar accuracy is achieved.

4.2.1.2 Bus fault injection

Compared to the processor model, fault injection in the system bus is considerably simplified due to its few communication states. The transmitted data is provided as an argument of the function for data transaction. Even without the API support, such transmitted data can be subjected to fault injection by changing its value at specific time instances directly. Similar to the processor models, faults are specified in the configuration file. During model initialization, the fault configuration is parsed and sorted into the fault queue according to their injection time. When data transport function is called, the scheduler detects whether a fault needs to be injected into the data before the actual data transmission according to the fault injection time.

4.2.1.3 Memory fault injection

Unlike processor and bus models, memory models in SystemC specification are not required to be clock-sensitive for a pure behavioral simulation. However, decision of fault injection needs to be made according to the current simulation time. For this purpose, an extra clock-sensitive SystemC method which maintains a clock counter is created for time-based fault injection, where the clock counter is continuously checked against the fault injection time. The proposed method can also be applied to other clock-insensitive modules. In the fault configuration phase, user needs to provide detailed memory array index for fault injection target or select a statistical distribution.

4.2.2 Experimental results

The Operating System Application Specific Instruction-set Processor (OSIP) [29] is a hardware accelerator of OS kernel providing support for task scheduling and synchronization in heterogeneous MPSoC. OSIP-based MPSoC system is used to verify the effects of system level fault injection. Modelled using Synopsys Platform Architect [1], the platform consists of seven ARM926EJ-S processors as PEs and one OSIP. Two applications on multi-processor have been investigated.

4.2.2.1 H.264 decoder

The application decodes H.264 data into video stream. ARM PEs dynamically get their task assignments from the OSIP. Fault configurations are parsed into the SystemC models of PEs. Note that currently no faults are injected into the OSIP. Figure 4.9 shows the impacts of faulty simulation compared with fault free simulation, where several effects are shown such as pixel error, thread error and fail to process.

Figure 4.9: H.264 decoder with fault injection

4.2.2.2 Median filter

Median filter implementation on OSIP reduces image noise by using PEs. Figure 4.10a) and 4.10b) show the original image with noise and the image after filtering respectively. A fault-tolerant implementation of median filter is executed on OSIP, which schedules additional tasks to other PEs whenever one PE is unresponsive. Downcount timers are implemented to restart the unresponsive PEs so that they get new tasks from the OSIP to continue processing. Currently the downcount time is set as $3\times$ of the regular processing time for one data token. Within such time if no pixel has been processed, the PE is considered to be unresponsive. Several experiments are conducted which are shown in Figure 4.11. The first experiment runs without fault injection, whereas for the rest experiments 100 bit-flip faults are injected on one PE. It can be observed from the Figure 4.11 that without monitoring timers, the processing overhead increases significantly as the number of hanging PEs grows. However, only a slight overhead occurs when timers are equipped, which is caused during task re-allocation.

Figure 4.10: Median filter: original and filtered image

Figure 4.11: Median filter: reliability exploration

4.2.3 Summary

A system-level fault injection technique for SoC components is proposed during the design of MPSoC. Faults on SystemC modules such as processor IPs, bus and memory can be efficiently injected. Using the proposed technique, system reliability is fast explored for applications running on multiprocessor system.

4.3 High-level Processor Power/Thermal/Delay Joint Modelling Framework

As reliability becomes an essential factor in the design of nanoscale digital system, it is important to integrate reliability as a design constraint in the traditional processor design flow, where instruction-set simulator plays an important role in architecture validation and performance estimation. The fault injection technique in Section 4.1 provides an user configurable approach to simulate processor behavior under fault. However, reliability effects, especially ageing and soft errors, have direct relationship with other design parameters such as runtime, power and temperature. There is strong need to link reliability with other physical metrics in a high-level processor design environment, where realistic estimation of reliability effects can be simulated together with power and thermal footprints.

Processor power estimation techniques have been continuously a hot topic in both research and industry. Instruction level power model is proposed by Tiwari et al. [187] [188], where each instruction is provided with an individual power model. The run-time power can be determined through the profiling of executed instructions. Wattch [41] introduces architecture-level power model which decompose main processor units into categories based on their structures, and separates each of the units into stages and forms RC circuits for each stage. McPAT [110] models all dynamic, static and short-circuit power while providing joint modelling capability of area and timing. To increase modelling accuracy, a hybrid FLPA(functional level power analysis) and ILPA(instruction level power analysis) model [24] is elaborated which advantageously combines the lower modelling and computational efforts of an FLPA model and the higher accuracy of an ILPA model. The trade-off is further explained in [139] with a 3-D LUT and a tripartite hyper-graph.

The heat dissipation from power consumption leads to increased and un-evenly distributed temperature which causes potential reliability problems [6] [96], where the research committee demands highly for architecture-level thermal management techniques. Consequently, accurate architecture level thermal modelling has received huge interests. In this domain, HotSpot [174] is the de facto standard, where the thermal effects for individual architecture blocks can be fast estimated. HotSpot is easy to integrate with any source level power simulator, which spreads its appliance into huge research bodies [84] [49].

Recently, there is an emerging research trend for multi-domain simulation, where physical factors in more than one system such as electrical, chemical and mechanical are jointly simulated [60]. In the domain of digital processor design, Cacti [136] estimates power, area and timing specifically for memory system. McPAT [110] jointly models power, area and timing for individual system-level blocks including cores and memories. [82] applies a joint performance, power and thermal simulation framework for the design of network-on-chip. [185] extends the work with the ability to simulate optimization techniques such as Dynamic Voltage Frequency Scaling (DVFS) and Power Gating.

However, the previous work simulates the physical behaviors using off the shelf libraries on a higher abstraction level for individual blocks, which did not deal with the complexity of processor architecture itself. An Application-specific Integrated Processor (ASIP) can have arbitrary logic blocks which need detailed block level modelling of physical parameters. Previous work also lacks the ability to accurately estimate power/temperature with application specific switching activities. The reason is that modelling and simulation are treated as separate issues, where the modelling part is more tent to be provided from IP vendors as technology dependent databases. Furthermore, to the best knowledge, no work has been ever attempted to integrate reliability issue directly into the joint simulation framework. Such issues still remain open to be addressed.

Contribution In this work, a joint modelling framework is demonstrated by integrating power, thermal and logic delay in a high-level processor design environment, where both accurately modelling through low-level characterization and cross-domain simulation using instruction-set simulator are fast realized. The reliability simulation is achieved as an extension to the high-level fault injection technique [209], where faults are modelled as delay variation on logic paths resulted from instantaneous power and thermal footprints. By automating the complete modelling and simulation flow, the processor designer can easily perform architectural and application-level design space exploration with power, temperature and reliability issues.

The work is organized in following manner. Section 4.3.1 discusses the approach of high-level power modelling and estimation for LISA based processor design framework. Section 4.3.2 illustrates the thermal modelling and integration using HotSpot package. Section 4.3.3 introduces the approach of high-level delay simulation. Section 4.3.4 focuses on the automation flow and analyses its runtime overhead.

4.3.1 High-level Power Modeling and Estimation

High-level power estimation flow characterizes and generates power models for LISA units (operations and resources) from low level power simulation. Such power models are applied later in instruction-set simulator to produce run-time power for targeted applications. The whole procedure is independent of PrimeTime based low-level power simulation so that significant less efforts are spent. The simulation accuracy depends on the efforts in power model characterization. This section covers the proposed flow. First, an overall introduction on the power modelling flow is presented. Second, the method of power model construction is discussed. After that, techniques to resolve power related factors such as inter-instruction effect are explained.

4.3.1.1 Flow Overview

Figure 4.12 explains the proposed power estimation flow which consists of simulation, characterization, estimation and exploration phases.

Figure 4.12: LISA-based power modelling and simulation flow

Simulation High level power models are usually characterized by the data from low level simulation. In this phase, cycle accurate power data for special testbenches are gathered according to PrimeTime-based power simulation which can be performed at either RTL, gate-level or layout. Currently gate level is chosen for trade-off between simulation accuracy and modelling efforts. Each simulation testbench consists of single type of instructions such as ALU, Load/Store and Branch types. The testbenches are composed in such a way that operands and immediate values are randomly distributed. Special instruction features such as operand bypassing are also covered according to the target architecture. Larger coverage of instruction modes and operand values leads to enhanced accuracy of power modelling.

Characterization Besides cycle accurate power data, the inputs of characterization phase also take into account the switching activities from instruction-set simulation. Both data are used to characterize the coefficients for the unit level power models which are detailed in the next section. Multivariate curve fitting technique is applied for extraction of coefficient values. Exploration between accuracy and extraction effort of coefficients can be explored through linear and polynomial modes in curve fitting.

Estimation The power models are applied on target applications to estimate power consumption. The run-time switching activities from cycle accurate instruction set simulator (ISS) are gathered and provided to the power simulator. Due to the nature

Figure 4.13: Hierarchical representation of RISC processor architecture

of unit-based power modelling, instantaneous power consumption for each hardware unit is calculated and recorded in the PrimeTime recognized power format.

Exploration The ISS-level run-time power is compared with low level simulation to determine the estimation accuracy. Based on the user requirements the unit-based power models can be further improved taking advantage of the techniques in the simulation and characterization phases.

4.3.1.2 Power Modelling

Finer granularity of power models from instruction level to architecture level enhances the power estimation accuracy. In an ADL-based processor simulation environment it is convenient to construct different levels of power models. An example is shown in Figure 4.13 where the hardware modules of a 5 pipeline stage RISC processor are listed in a hierarchical manner. The single module on level 0 represents the processor core. The instruction level power model can be created by power profiling and characterization according to different execution states/instructions of the processor core.

$$\hat{P} = \alpha_0 + \alpha_1 HD_1 + \alpha_2 HD_2 + \dots + \alpha_i HD_i$$

$\hat{P}$: estimated power, HD_i : Hamming Distance for signal i ,
 α_i : Power coefficient for signal i , α_0 : Base power coefficient

Figure 4.14: Unit-level power model

Architectural power model for level 1 and level 2 hardware modules are constructed depending on the granularity of the components. The proposed unit-based flow further expands logic modules on level 2 into logic operations listed in level 3 while keeps the storage resources such as pipeline registers and register file unexpanded. However, without module expansion, high level processor which has significant differences in runtime power consumption due to internal signal switching activities are extremely difficult to model accurately. A case of this is the inter-instruction effect. The advantage of such unit-based power model also lies in its ability in modelling generic architectures since all processor modules can be decomposed into several operation units and storage resources which can be individually characterized.

The concept of unit-based power model can be viewed from Figure 4.14 where the block represents either a logic operation or a storage cell. Read and write signals are connected to the unit which charge and discharge unit capacitance. The RTL equivalent names of read/write signals are automatically acquired from the resource table during model compilation. In Synopsys Processor Designer, signal names outputted from RTL generator are uniquely defined by the unit name, access resource name and

Figure 4.15: Processor resource table for HDL generation

access types (read or write). It is indicated in Figure 4.15, where a RISC processor is used to illustrate the resource table. For multiple similar accesses of same type within same logic unit, the names are resolved using sequential index numbers as suffixes. Further information on resource generation is explained in [2].

The power model for each unit is constructed using the toggling of extracted read and write signals which are represented by their *Hamming Distances* between clock cycles. The run-time power of individual unit is modelled based on the weighted summation of the power contributed from individual signals, where the weights of signals are extracted from low level power simulations in the form of coefficients by using characterization testbenches. Cycle accurate power consumption and signal hamming distances from the simulation phase are used to extract such coefficients.

Figure 4.14 also shows an example of coefficient extraction using interpolation techniques [99] for single coefficient. Both linear and polynomial fitted curves are indicated with their formula. This process is automatically performed to characterize all processor units. It is obvious that the order of the formula directly relates to its estimation accuracy, which also trades off for the timing overhead. In this work, linear curve fitting is applied to acquire all power coefficients.

However, the model constructed from weighted hamming distances is insufficient for accurately characterizing power consumption of hardware units. For logic operations one significant factor is the activation conditions. An inactivated logic operation

Figure 4.16: Separate modes for power models

with same hamming distances of signals consumes one order lower power than the activated operation since no logic operations are actually performed inside the operation. With regard to registers, two modes also need to be separately addressed for resource write and read. Gate-level simulation shows that the write causes one order higher power consumption than read on the registers. Such effects are addressed by modelling both operations and registers by using two power models which is shown in Figure 4.16. Based on the activation situation of operation units, the power simulator selects either activate or inactivate model for power calculation. The power value of registers is always the addition of both write and read power.

4.3.1.3 Power Related Factors

Inter-instruction Effect One key challenge for the accuracy of instruction level power modelling is the inter-instruction effect (IIE), where the power consumption caused by instruction transition varies among different instruction pairs. Previous work tries to characterize power consumption for all instruction pairs in the processor ISA which requires significant efforts. Besides, the difficulty of pair modelling scales exponentially with the increasing instruction set. An example is VLIW processor where instructions usually consist of several slots which increase the number of instruction pairs even more.

The unit level power model addresses the IIE problem from finer architectural granularity. The instruction transition is investigated through signal transitions for different units. Connecting instruction pairs would possibly activate different logic operations in each pipeline stage. Since the instruction level power model already considers the activation power of current operations, the IIE power is the deactivating power for previous operations which can be derived from the inactivate mode of operation power models. For registers, the read and write power are calculated in each clock cycle to cover the resource transition power including IIE power.

Custom Instructions For a processor with N instructions, a new implemented custom instruction which adds one unit in the architecture would increase the number of instruction pairs by $2N$. However, only small amount of additional units need to be characterized in the proposed flow, which demonstrates the flexibility of unit power model compared with instruction level power models.

4.3.1.4 Experimental Results in Power Modeling

Instruction-level Power The target RISC processor is synthesized at 500MHz under 90nm technologies. To evaluate the accuracy, firstly specific testbenches used in power characterization are considered. Average error of power estimated by proposed power models and that with PrimeTime are calculated and the results are summarized in Table 4.5. Except *ld_rr*, power difference for all instruction groups are below 5%. Power for memory loading is relatively inaccurate with a difference of 7.89% because of the limited information about unit MemoryFile at the LISA level. Besides, MemoryFile unit implements the memory interface where the actual memory implementation is not characterized currently.

Table 4.5: Power estimation accuracy for each instruction group

Instruction	Difference	Instruction	Difference
alu_rri	2.55%	cmp_ri	3.03%
alu_rrrr	0.36%	cmp_rr	3.63%
alu_rrri	1.62%	st_rr	2.86%
alu_rrr	1.99%	ld_rr	7.89%
ldc_ri	3.48%	bra	2.28%
lui_ri	3.64%	brau	0.53%

Instruction-level power models are easy to extend to other processors. Power models for another RISC processor with mixed 16/32 bits ISA under four technologies are further characterized using testbenches with random operators. The processor is synthesized at 25MHz. Figure 4.17 presents the instruction level power consumption.

Application-level Power Six embedded applications are simulated to test the power estimation flow. The application profiling based on instruction groups are shown in top half of Figure 4.18 while the comparison for average power and simulation speed is documented in bottom half. The LISA-based power simulation results show a close approximation to the PrimeTime-based simulation. The relative higher error for Sieve application results from its large amount of memory load instructions which can be observed by application profiling. This can be reasoned from table 4.5 where the memory loading leads to higher inaccuracies than other instructions. In terms of power simulation speed, current implementation achieves in average 28x faster

Figure 4.17: Instruction-level power for RISC processor

for target applications compared with gate-level simulation. The factors for LISA-level power simulation overhead include intensive calculation for signal hamming distances, power coefficients addressing and the dumping of power data. Noted that high-level power simulation has its limitation in reducing the accuracy gap, which is caused by the significantly reduced number of signals compared with gate-level simulation. Consequently, the contribution of power consumption from many gate-level signals can not be tracked in high-level simulator, which reveals the trade-off between accuracy and simulation speed.

The snapshots of instantaneous power consumption for high-level and PrimeTime simulation for selected applications are shown in Figure 4.19. The waveforms show a good run-time power estimation which implies the usage of proposed flow for application and architecture level power analysis.

Power for Custom Instruction To demonstrate the flexibility of the proposed flow for modelling custom instructions, the Zero Overhead Loop (ZOL) instruction is implemented for the RISC processor. ZOL is used to execute a number of subsequent instructions for specified number of iterations. In LISA model, one additional *ZOL_control* operation for setting program counter based on the loop count value is created whereas the *FETCH* operation is updated for checking the ZOL branch conditions. Power modelling for ZOL instruction targets on the characterization of *ZOL_control* and the updated *FETCH* operations based on the testbenches. Comparison between LISA level power estimation and PrimeTime in Table 4.6 shows a modelling error of 3.63% with the model characterization time of around 10 minutes.

Figure 4.18: Application profiling and average power

Table 4.6: Power estimation for custom instruction

Instruction	LISA-based power	Gate-level power	Difference
ZOL	5.1mw	5.3mw	3.63%

Figure 4.19: Instantaneous power for selected applications

4.3.2 LISA-based Thermal Modeling

4.3.2.1 Thermal Modelling using HotSpot

HotSpot is an opensource package for temperature estimation of architecture-level units. It has been applied in both academia and industry for architecture-level thermal modelling and management. HotSpot is easily integrated into any performance/power simulator by providing the floorplan and instantaneous power information. By transforming the floorplan into an equivalent thermal RC circuits which is called *compact models*, HotSpot calculates instantaneous temperature by solving the thermal differential equation using a fourth-order Runge-Kutta method. The temperature for each block is updated by each call to the RC solver. For details of applying HotSpot for thermal modelling please refer to [84].

4.3.2.2 Integration of Power Simulator with HotSpot

The integration of LISA power simulator with HotSpot generally follows the guideline in [81]. Two phases are required, the initialization and runtime phases, which are briefly explained in the following.

```
# Line Format: <unit-name>\t<width>\t<height>\t<left-x>\t<bottom-y>
# all dimensions are in meters
# comment lines begin with a '#'

ARC_FE          0.0001756      0.0000446      0.0020560      0.0010420
ARC_DC          0.0011928      0.0010360      0.0013692      0.0004060
ARC_EX          0.0010528      0.0014280      0.0002688      0.0003780
ARC_MEM         0.0005264      0.0005320      0.0020300      0.0014700
ARC_WB          0.0006804      0.0003920      0.0013384      0.0004700
ARC_FE_DC       0.0012400      0.0000659      0.0001704      0.0000712
ARC_DC_EX       0.0009096      0.0007560      0.0002912      0.0018340
ARC_EX_MEM      0.0006832      0.0007000      0.0012908      0.0018900
ARC_MEM_WB      0.0004480      0.0005898      0.0021112      0.0020020
ARC_RegisterFile 0.0012400      0.0014042      0.0020560      0.0005260
```

Figure 4.20: Floorplan information for input of HotSpot framework

- The initialization phase, where the RC equivalent circuits are first initialized based on user provided floorplan and thermal configurations, such as parameters for heat sink and heat spread. Afterwards, the initial temperature is set by the user. For instance, 60 degree is initialized for starting temperature while 45 degree is set as ambient temperature. Figure 4.20 shows an example of the floorplan information of a RISC processor, which contains 5 pipeline stages and 4 stages of pipeline registers. Such file could be obtained from commercial physical synthesis tool such as Cadence SoC Encounter or derived according to the area report from logic synthesizer. The data in Figure 4.20 are calculated according to the area of individual architectural units.
- The runtime phase, where the simulator iteratively calls the temperature computing routine to update the block temperatures. Such routine does not need to be called during each clock cycle due to the nature of slow changing temperature. In practice, a sampling interval of 10 Kilocycles at 3 GHz is adapted, which corresponds a time of 3.33 microseconds. For different clock frequency, the same interval is maintained to make fair comparison. The power values which provide to HotSpot are the average values among the previous sampling interval.

4.3.2.3 Temperature Simulation and Analysis

Figure 4.21 shows an example of the runtime temperature simulation for BCH application under a synthesis frequency of 500 MHz. The unit of time is in nanosecond while the temperature is in Degree Celsius.

Table 4.7 shows the temperature and power consumption for architectural units with different design frequencies, where BCH application runs on the processor. The same floorplan as in Figure 4.20 is applied for all frequencies. As the power increases dramatically with frequency, the temperature shows slightly increment for most of the

Time (nanosec)	Temperature (Degree Celsius)									
	FE	DC	EX	MEM	WB	FE_DC	DC_EX	EX_MEM	MEM_WB	RegisterFile
3320	60.01	60.00	60.00	60.00	60.00	60.01	60.00	60.00	60.00	60.01
6680	60.03	60.00	60.00	60.00	60.00	60.03	60.01	60.01	60.01	60.01
10040	60.04	60.00	60.00	60.00	60.00	60.04	60.01	60.01	60.01	60.02
13400	60.06	60.00	60.01	60.00	60.00	60.05	60.01	60.01	60.01	60.02
16760	60.07	60.00	60.01	60.00	60.00	60.06	60.02	60.02	60.02	60.03
20120	60.09	60.00	60.01	60.00	60.00	60.07	60.02	60.02	60.02	60.03
23480	60.10	60.01	60.01	60.01	60.00	60.08	60.02	60.02	60.02	60.03
26840	60.12	60.01	60.01	60.01	60.00	60.09	60.02	60.03	60.03	60.04
30200	60.13	60.01	60.01	60.01	60.00	60.10	60.03	60.03	60.03	60.04
33560	60.14	60.01	60.01	60.01	60.00	60.11	60.03	60.03	60.03	60.05
36920	60.16	60.01	60.01	60.01	60.00	60.12	60.03	60.03	60.04	60.05
40280	60.17	60.01	60.02	60.01	60.01	60.13	60.04	60.04	60.04	60.06
43640	60.18	60.01	60.02	60.01	60.01	60.14	60.04	60.04	60.04	60.06
47000	60.19	60.01	60.02	60.01	60.01	60.15	60.04	60.04	60.04	60.07
50360	60.21	60.01	60.02	60.01	60.01	60.16	60.04	60.05	60.05	60.07
53720	60.22	60.01	60.02	60.01	60.01	60.17	60.05	60.05	60.05	60.07
57080	60.23	60.01	60.02	60.01	60.01	60.18	60.05	60.05	60.05	60.08
60440	60.24	60.01	60.02	60.02	60.01	60.19	60.05	60.05	60.06	60.08
63800	60.26	60.01	60.02	60.02	60.01	60.20	60.05	60.06	60.06	60.09
67160	60.27	60.01	60.03	60.02	60.01	60.21	60.06	60.06	60.06	60.09
70520	60.28	60.01	60.03	60.02	60.01	60.22	60.06	60.06	60.07	60.09
73880	60.29	60.02	60.03	60.02	60.01	60.23	60.06	60.06	60.07	60.10
77240	60.31	60.02	60.03	60.02	60.01	60.24	60.06	60.07	60.07	60.10
80600	60.32	60.02	60.03	60.02	60.01	60.25	60.07	60.07	60.07	60.11
83960	60.33	60.02	60.03	60.02	60.01	60.26	60.07	60.07	60.08	60.11
87320	60.34	60.02	60.03	60.02	60.01	60.27	60.07	60.07	60.08	60.11
90680	60.35	60.02	60.03	60.02	60.01	60.28	60.07	60.08	60.08	60.12
94040	60.36	60.02	60.03	60.02	60.01	60.29	60.08	60.08	60.08	60.12
97400	60.38	60.02	60.04	60.02	60.01	60.30	60.08	60.08	60.09	60.13
100760	60.39	60.02	60.04	60.03	60.01	60.30	60.08	60.08	60.09	60.13

Figure 4.21: Instantaneous temperature generated by HotSpot

units such as DC, MEM and WB. Rapid increment lies between 100MHz and 500MHz for units FE and FE_DC, even though their power consumptions are relatively small compared with other units. On the contrary, units such as RegisterFile which incur higher power consumption shows only a slight increment in temperature. The reason behind this is the high power density on such units due to their small area provided by the floorplan.

Table 4.7 shows the temperature for BCH application using different floorplans. The first floorplan as in Figure 4.20 adopts the ratio of unit size from logic synthesis tools. However, the runtime temperature shows strong differences among different architectural units, which has the potential to incur temperature related reliability issues. Floorplan 2 tries to increase the sizes of units with high power density so that the power density will be significantly reduced. As seen from the thermal simulation, the temperature of hot units reduces dramatically so that the thermal footprints of pipeline registers and RegisterFile are finalizing at similar values. To prevent large area overhead, a slight increment to the area of registers is introduced due to their initially large size. The area of FE is also increased to achieve uniform temperature for all logic between pipeline stages. Overall a 38.6% area overhead is incurred to achieve the thermal footprints where all units show temperature under 68 degree. In other words, it reflects a maximal power density around $2.00 \text{ W}/\text{m}^2$ for arbitrary

Frequencies	25MHz		100MHz		500MHz	
Units	temp($^{\circ}$ C)	power(mW)	temp($^{\circ}$ C)	power(mW)	temp($^{\circ}$ C)	power(mW)
FE	63.90	3.19e-3	69.39	9.08e-3	84.25	3.88e-2
DC	60.16	2.56e-2	60.43	7.18e-2	61.15	2.99e-1
EX	60.23	4.91e-2	60.60	1.40e-1	62.11	7.06e-1
MEM	60.17	3.88e-3	60.63	9.53e-3	61.48	3.72e-2
WB	60.05	2.69e-3	60.20	8.47e-3	60.66	3.86e-2
FE_DC	62.16	2.71e-2	68.44	1.05e-1	89.04	4.93e-1
DC_EX	60.74	5.72e-2	62.66	2.08e-1	67.68	1.08
EX_MEM	60.74	4.09e-2	62.40	1.50e-1	67.66	7.61e-1
MEM_WB	60.45	2.32e-2	61.74	8.73e-2	66.89	4.36e-1
RegisterFile	61.09	2.39e-1	63.25	7.54e-1	69.79	3.52

Table 4.7: Temperature and power of LT_RISC at different frequencies running BCH application

Units	Power @500MHz (mW)	Floor plan 1			Floorplan 2		
		size (mm^2)	power density (W/m^2)	temp ($^{\circ}$ C)	size (mm^2)	power density (W/m^2)	temp ($^{\circ}$ C)
FE	3.88e-2	0.01	4.95	84.25	1.00	0.04	60.19
DC	2.99e-1	1.24	0.24	61.15	1.24	0.24	61.15
EX	7.06e-1	1.50	0.47	62.11	1.50	0.47	62.11
MEM	3.72e-2	0.28	0.13	61.48	0.28	0.13	61.37
WB	3.86e-2	0.27	0.14	60.66	0.27	0.14	60.66
FE_DC	4.93e-1	0.08	6.03	89.04	1.00	0.49	62.40
DC_EX	1.08	0.69	1.57	67.68	0.76	1.43	67.02
EX_MEM	7.61e-1	0.48	1.59	67.66	0.49	1.55	67.42
MEM_WB	4.36e-1	0.26	1.65	66.89	0.30	1.45	66.17
RegisterFile	3.52	1.74	3.52	69.79	2.25	2.02	67.57
Total	-	6.55	-	-	9.08	-	-

Table 4.8: Temperature of LT_RISC running BCH application using different floor-plans

logic units. According to the strong relationship of temperature with power density, further thermal optimization techniques could be purposed.

Table 4.9 shows the temperature of processor units by end of the simulation time for 10 embedded benchmarks using the initial floorplan in Figure 4.20. The temperature differs among applications mainly due to the difference in execution time of the applications. For instance the BCH application which runs for 900 μ s is significantly hotter on most of the units than other short applications. For applications with similar execution time such as CRC32 and Sieve, no huge differences in temperature among all units is detected. Note that change in temperature is a slow process compared with power consumption, where application dependent thermal effects will exhibit for long execution time. For instance, with 91.4% execution time of median application, viterbi achieves a slightly higher temperature in EX units, which is due to the nature of more ALU instructions. Assembly level profiling shows that viterbi incurs

Units	Temperature ($^{\circ}\text{C}$) for different applications									
	bch	cordic	crc32	fft	idct	median	qsort	sieve	sobel	viterbi
FE	84.25	60.38	61.10	60.57	61.70	73.67	72.62	61.27	60.73	72.65
DC	61.15	60.02	60.06	60.03	60.09	60.70	60.65	60.06	60.04	60.69
EX	62.11	60.05	60.15	60.06	60.20	61.50	61.51	60.07	60.10	61.69
MEM	61.48	60.03	60.07	60.03	60.09	60.94	60.95	60.02	60.04	60.80
WB	60.66	60.02	60.05	60.02	60.06	60.50	60.52	60.03	60.03	60.48
FE_DC	89.04	60.44	61.29	60.66	62.04	76.01	75.13	61.51	60.86	75.50
DC_EX	67.68	60.12	60.34	60.17	60.55	64.25	64.02	60.37	60.23	64.06
EX_MEM	67.66	60.12	60.36	60.18	60.54	64.32	64.14	60.38	60.25	64.22
MEM_WB	66.89	60.14	60.39	60.19	60.57	64.42	64.31	60.39	60.27	64.22
RegisterFile	69.79	60.15	60.44	60.22	60.70	65.33	65.17	60.48	60.30	65.10
Finish time (μs)	900.2	6.7	20.0	10.0	33.3	350.1	333.4	23.3	13.3	320.1

Table 4.9: Temperature of LT_RISC at 500MHz for different applications

59,739 ALU instructions (37.12% of all instructions) while median has the amount of 46,301 (26.39% of all instructions), which verifies viterbi's hotter temperature in EX pipeline unit than that for median.

4.3.3 Thermal-aware Delay Simulation

The effects of temperature on the logic delay of nanoscale CMOS technology have been heavily investigated such as Negative-bias Temperature Instability (NBTI) [6] and Inverted Temperature Dependence (ITD) [96]. Most of previous work focus on device and gate-level. Such effects can be modelled using the architectural level thermal simulation framework proposed in this work, so that an thermal-delay simulator for generic processor architecture could be easily generated and explored.

Figure 4.22 shows the integration framework with power and thermal simulator to model the delay fault. As discussed in Section 4.3.2, the LISA-level temperature simulator is generated using power simulator, HotSpot package and architectural floorplan. Thermal directed delay fault is modelled by combining the thermal simulator and the high-level timing fault injection discussed in Section 4.1.3, where the runtime delay of individual logic paths is updated using temperature and a user provided delay variation model. In this section, the effects of delay change with temperature are modelled according to a second order polynomial model for 65nm technology. The effect of ITD for different applications running on a RISC processor is also presented.

4.3.3.1 Inverted Temperature Dependence

Propagation delay of CMOS transistor is widely modelled using the *Alpha-power* law [163] as:

$$\text{Delay} \propto \frac{C_{out} V_{dd}}{I_d} = \frac{C_{out} V_{dd}}{\mu(T)(V_{dd} - V_{th}(T))^\alpha} \quad (4.2)$$

Figure 4.22: Thermal-aware fault injection

where C_{out} is the load capacitance, α is a constant, $\mu(T)$ is the temperature dependent carrier mobility, $V_{th}(T)$ is the temperature dependent threshold voltage. The temperature affects the delay in two ways: at high voltage V_{dd} , delay is less sensitive to the term $V_{th}(T)$ but to the mobility, while at low temperature the thermal effects on threshold voltage dominates the delay change. As a consequence, for advanced technology which has small driving voltage, the increment in temperature could reduce the propagation delay rather than increase it for technologies with higher voltage. Such effect is named as *Inverted Temperature Dependence* (ITD) and the voltage which inverts the trend of thermal dependent, is the *Zero-temperature coefficient* (ZTC) voltage.

4.3.3.2 Timing Variation Function for Inverted Temperature Dependence

The effects of ITD for 65nm technology are modelled using the trend of delay change for clock tree network in [165]. Two assumptions are made to simplify the high-level modelling:

1. The delay of logic path follows the same ration of temperature/voltage dependency of individual logic buffer.
2. The temperature within one architecture block is uniform.
3. Other thermal effects on the change of threshold voltage such as NBTI is not modelled currently.

Figure 4.23 shows two critical paths for the RISC processor and their transverse architectural blocks, which are generated by the STA tools. The delay of the complete logic path equals to the sum of path delay of individual logic blocks on the path. For instance, the critical path 1 which gets two operands from pipeline register and RegisterFile transverse in order the following block: *MEM_WB*, *DC*, *BYPASS_DC*, *DC*,

Figure 4.23: Critical paths and transverse blocks

RegisterFile, *DC*, *ALU_DC*, *DC*, *DC_EX*. The critical path 2 transverses *EX_MEM*, *EX*, *BYPASS_EX*, *EX*, *ALU_EX*, *EX*, *EX_MEM*. The delay within individual architectural units are updated using its own running temperature, which is generated from the thermal simulation. In extreme case, each cell uses its own running thermal footprints to update its delay, which can only be simulated using gate-level thermal analysis.

With the above assumption and the referred data for 65nm technology in [165], the second order polynomials shown in Figure 4.24 are interpolated to represent the relationship between supply voltage, instantaneous temperature and propagation delay.

It is observed that the trend of propagation delay with temperature differs with supply voltage. For 1.0V and 1.1V the delay increases with temperature while decreases at 0.9V. In [165] the ZTC voltage is known to be 0.95 V for 65nm technology from STMicroelectronics, which proves the effect of ITD for advanced technology.

4.3.3.3 Case Study for ITD Simulation

The polynomials are used as the path timing variation models for the RISC processor and to test the change of critical path running embedded applications. Figure 4.25 shows the runtime delay of the critical path for the RISC processor running BCH application. Curves are plotted for both frequency of 25MHz and 500MHz. The supply voltage is simulated using 0.9V, 0.95V, 1.0V and 1.1V. The initial delay of critical path extracted out of the timing analysis tool is for the worst case condition under 125⁰C,

Figure 4.24: Delay variation function under several conditions

Figure 4.25: Runtime delay of critical path for BCH application

Figure 4.26: Automation flow of power/thermal/logic delay co-simulation

0.9V. It is observed that for high supply voltage such as 1.1V and 1.0V, the delay increases with temperature till a saturation point then slightly decreases according to the characteristics of the application. For low voltage of 0.9V, the inverse trend is shown where the delay decreases with temperature till the saturation point and then slightly increases. Under the ZTC voltage which is 0.95V, the delay is not affected by the temperature as expected. The effect of ITD shows potential of frequency overscaling under lower voltage, which is predicted for 65nm and further technologies in [213]. With regard to different running frequencies, the processor running at 500MHz consumes higher power which leads to higher temperature compared to the data at 25MHz. Consequently, the speed of delay change shows more significant dependence on temperature for higher frequencies.

4.3.4 Automation Flow and Overhead Analysis

In this section the purposed automated estimation flow for Power/Thermal/Delay is briefly documented, which functions as a simulator wrapper to the Synopsys Processor Designer [182]. Furthermore, the overheads for both characterization and simulation are discussed.

4.3.4.1 Flow Summary

Figure 4.26 illustrates the complete analysis framework, where the architecture description and application of interests are provided as inputs. The framework consists of characterization and simulation phase. The power characterization phase consists of 4 modules, which are briefly explained:

Testbench generation is used to generate processor specific testbenches for power characterization. This module parses the syntax section of processor description to produce instructions with random operands. One testbench is generated for each type of instruction, which runs for a predefined simulation clock cycles.

Resource table extraction gets the hierarchical information of the architecture and extracts input and output signals for each architecture unit. Read and write power models in the form of interpolated polynomial will be generated to each unit.

Behavioral simulation dumps the runtime hamming distance of input/output signals per architecture unit, which is used for power coefficient extraction.

Power LUT extraction interpolates power coefficients in the form of LUT using hamming distance and data from low-level power simulation. The interpolation itself is carried out using Matlab tool.

Power simulation takes loops to simulate processor behavior and power consumption until end of the simulation cycles. In each control step the simulator calculates power consumption based on the architecture unit specific instruction type, runtime hamming distances of the pins and power coefficient of the architecture units. Instead of list based implementation of power LUT, hash container is applied to increase the speed of instruction-architecture specific LUT addressing. The hierarchical power data according to Figure 4.13 is dumped during simulation. More modelling architecture units lead to higher overhead of power estimation.

Thermal and delay simulations are automatically generated once upon power simulator is ready, since no further characterization steps is required for thermal and delay simulation.

The proposed flow is demonstrated by using Synopsys Processor Designer and is portable to any high-level architecture simulation environment and architectures. Further work includes the porting of the framework into other ADL such as SystemC.

4.3.4.2 Overhead Analysis

Table 4.10 shows the timing and accuracy for power characterization phase under two groups of testbenches, where 10 architecture units are modelled. The first group consists of 14 types of instructions to cover the most generalized processor instructions. For instance, ALU instructions such as add, sub and and which operate on 2 register operands and 1 immediate are grouped together in one instruction type. The second group consists of 33 types of instructions where each instruction type consists of exact one operational mode. The characterization is performed on the machine with Intel Core i7 CPU at 2.8 GHz. Each instruction file is running for 2,000 clock cycle.

As shown in the Table 4.10, group one achieves faster characterization time than group two. However, group two achieves higher estimation accuracy when benchmarked with gate-level power estimation. Generally, the power characterization time in the range of several minutes is acceptable for power modelling of embedded processors.

Number of testbenches	14 instructions	33 instructions
Time (minutes)	3	8
Average error (%)	21.3	8.6

Table 4.10: Time and accuracy of power characterization for testbench groups

Table 4.11 represents the runtime overhead of different simulation mode including pure behavioral simulation, power estimation, thermal estimation and delay simulation, where 10 architecture units are modelled. It is observed that the runtime overhead significantly lies in the power estimation compared with behavioral simulation, on which details have been discussed in Section 4.3.1.4. The thermal simulator achieves only 1.2% of overhead compared with power simulator, which is due to the light weight implementation of HotSpot package and smooth integration with power simulator. The delay simulation achieves in average 6.3% of overhead compared with thermal simulator, which is mainly due to the parsing of delay information from timing analysis file which contains delay of the longest 1,000 paths.

Simulator Applications	Behavior (sec)	Power (sec)	times	Thermal (sec)	+%	Delay (sec)	+%
BCH	2.04	124.94	61x	125.47	0.4	129.72	3.4
Viterbi	0.82	43.49	53x	44.37	2.0	47.86	7.9
Median	0.87	49.40	57x	49.45	0.1	53.00	7.2
Qsort	0.81	45.45	56x	46.65	2.6	48.53	4.0
IDCT	0.19	5.17	27x	5.22	1.0	5.69	9.0
Average	-	-	51x	-	1.2	-	6.3

Table 4.11: Runtime overhead for different simulation modes

4.3.5 Summary

In this work, a processor power/thermal/delay joint modelling framework is presented for ADL LISA-based processor design environment. Detailed experiments are conducted which explore the usability of the framework with several design parameters such as applications, technologies and layouts. An automatic setup has been constructed which performs estimation and analysis according to such parameters.

The proposed framework helps processor designer to explore the physical effects in early design stage.

Chapter 5

Architectural Reliability Estimation

In this chapter three reliability estimation techniques are illustrated which fast characterize the effects of errors on processor architecture. In Section 5.1 an analytical estimation technique is presented to quantify the vulnerability and logic masking capability of individual circuit elements while calculating instruction and application level error rates. In Section 5.2 Probabilistic Error Masking Matrice is introduced to predict error effects through the graph network of dynamic processor behavior. In Section 5.3 design diversity metric is illustrated to evaluate the robustness of redundant system against common mode failures for system-level processing components.

5.1 Analytical Reliability Estimation Technique

Complementing the simulation techniques using fault injection, analytical techniques have also been proposed to investigate behavior of circuits under faults. Mukherjee et al. [135] introduced the concept of architecturally correct execution (ACE) to compute the vulnerability factors of faulty structures. In [23] the authors performed the ACE analysis to compute architectural vulnerability factors for cache and buffers. Recently, Rehman et. al [150,162] extended the ACE concepts to instruction vulnerability analysis and proposed reliability-aware software transformations. The vulnerability of the instruction is analysed in this work by studying the constituent logic blocks and possibly connect with the circuit-level reliability analysis [160]. While the instruction vulnerability index model proposed at [150] includes the logical masking effects, the details of derivation of the masking effect are not mentioned. The simulation accuracy is compared with other software-level reliability estimation flows [150].

Contribution In this work, an analytical technique is proposed to estimate the application dependent reliability of embedded processors and benchmark its usage on fault evaluation with instruction set simulation-based fault simulation technique in Section 4.1. Figure 5.1 shows the contributions where the novel modules are filled in dark color. The simulation-based reliability estimation technique is performed for both RTL and ADL abstraction layers. The analytical technique takes the instruction profiling of the target application and fault simulation results at either abstraction layer as inputs. Such results are used to calculate the operation fault properties and Instruction Error Rate (IER) which are then processed by the reliability estimator to predict the Application Error Rate (AER). Users can improve LISA models and target applications to tune the AER, which closes the reliability estimation/exploration loop.

Figure 5.1: ADL driven reliability estimation flow

To present the analytical technique, the operation reliability model is explained first, which is applied in the following to calculate instruction error rate. Then the application error rates are derived by profiling the target applications. The exemplary analysis is carried on the 5-pipeline stages RISC processor model, which is available via [182].

5.1.1 Operation Reliability Model

Directed Acyclic Graph (DAG) is used to represent the activation chain of LISA operations. To represent fault injection and error propagation, data flows have to be added in the DAG. Figure 5.2 shows the data flow graph for the ALU instruction. While the nodes represent LISA operations the edge between them shows the data flow with an individual index and corresponding signal names. When a transient fault is injected into an operation, it needs to first manifest on the operation's output edges and then propagate through following operations until it manifests on the output of the Writeback operation to result in an instruction level error. Notice that not all faults will result in an instruction level error due to logic masking effect. Consequently,

the operation error probability and masking probability are proposed to model such process.

Figure 5.2: Data flow graph for ALU instruction

Operation error probability C_{op}^e is the probability of a detected error on the output edge e of an operation when a fault is injected inside its operation.

Operation masking probability $M_{op}^{e_{in}, e_{out}}$ is the probability of a detected error on the output edge e_{out} of an operation when a fault is injected in its input edge e_{in} .

Each operation has both C_{op}^e and $M_{op}^{e_{in}, e_{out}}$ to represent the situation of fault injection on it and error propagation through it respectively. For a particular architecture model, single bit fault is injected through disturbance signals inside of each operation randomly in time and location. By tracing the output edges and comparing the traced value with golden simulation, it is easy to get C_{op}^e when large number of simulations are performed to counter the randomness. $M_{op}^{e_{in}, e_{out}}$ can also be acquired when faults are injected to the input edges while output edges are traced and compared. Pure analysis on the data flow graph of combinational logic inside each operation instead of simulation method can also predict its C_{op}^e and $M_{op}^{e_{in}, e_{out}}$ value, which will be proposed in the future work.

5.1.2 Instruction Error Rate

The path error probability is the product of C_{op}^e and the $M_{op}^{e_{in}, e_{out}}$ of its following operations on the same path from the fault injected operation to the sink operation.

The instruction error rate $IER_{insn}^{op_faulty}$ for operation op_faulty and for instruction $insn$ is defined as the summation of all path error probabilities. For example Equation 5.1 shows the instruction error rate when operation Fetch in Figure 5.2 is fault injected. The edges in the equation are labelled by their indexes.

$$IER_{alu}^{fetch} = C_{fetch}^1 M_{decode}^{1,2} M_{writeback}^{2,7} + C_{fetch}^1 M_{decode}^{1,3} M_{alu_ex}^{3,6} M_{writeback}^{6,7} + C_{fetch}^1 M_{decode}^{1,4} M_{alu_dc}^{4,5} M_{alu_ex}^{5,6} M_{writeback}^{6,7} \quad (5.1)$$

The method above to calculate the instruction error rate can be applied to all instructions which are defined as a chain of activated operations in LISA. An instruction error can be resulted from a fault injected in each preceding operation in the instruction data flow graph. So that the error rates for a particular instruction constitute a set of $IER_{insn}^{op_faulty}$ where op_faulty is one of the activated operations for $insn$. Besides the operations the edges between them can also be faulty, which resembles the situation when fault is injected on storage resources such as signals and registers. Such resources have essentially both error and masking probabilities equal to one since no masking effect exist for the resources, so that they propagate any encountering fault. In this work the SEU errors caused within the resources are not considered since the analysis primarily is focused on those caused inside combinational logic.

5.1.3 Application Error Rate

The application error rate $AER_{app}^{op_faulty}$ represents the error probability when a fault is injected inside operation op_faulty during the execution of a specific application app . When the error rates for all the instructions are known, the application error rate is defined to be the weighted average of all instruction error rates, where the weight of each instruction is its execution counts versus the total instruction counts of the whole application. Figure 5.3 shows the DAG for all instructions of the RISC processor model. Several instructions which have similar operand behaviors are grouped into the same operation for simplicity. Each instruction corresponds to a path starting from operation Fetch to its sink operations, which interact with resources such as register file or memories. The weights of instructions are labelled as pi , which can be acquired from the application profiler. As an example, the application error rate of alu_rrr_ex operation is shown in Equation 5.2. The summation happens since the operation is on the activation chain of two instructions alu_rrr and alu_rrri .

$$AER_{app}^{alu_rrr_ex} = p_{app}^1 IER_{alu_rrri}^{alu_rrr_ex} + p_{app}^2 IER_{alu_rrr}^{alu_rrr_ex} \quad (5.2)$$

The application error here is detected through the mismatch of instruction results, either committed values to register files or load/store values to memories, with the golden simulation. This provides a conservative estimate of the error rate in program's output, which is normally the value sent by the processor through I/O instruc-

tions. The error in the current setup may not lead to an I/O error. This can be caused by several factors. First, the erroneous value committed to architecture registers can be masked by following instructions before I/O access. Second, affected operations which are not activated can be irrelevant to the value finally sent through I/O. Besides, the hardware bypass features in the processor can also silent the interface error since the source of operands can be the bypassed value from pipeline registers instead of architecture registers. In this case an error occurring at the writeback value after it is bypassed to later instructions may also not result in an error. However, the proposed analysis offers a fast method to determine to what extent a fault injected operation can potentially influence the program output so that engineers can adopt software or hardware measures to improve system reliability.

Figure 5.3: Operation graph for all instructions in RISC processor

5.1.4 Analytical Reliability Estimation for RISC Processor

In this section, the reliability analysis based on the proposed methodology is presented. First, the estimation of *IER* for individual operation is shown. In the next *AER* is calculated from *IER* and application dependent weights of instructions. The estimated values are compared with experimental values.

5.1.4.1 IER

A set of testbenches are developed to get individual $IER_{insn}^{op_faulty}$. Each testbench contains the same type of instructions with different modes and random operands. Single bit-flip fault with duration 1 clock cycle targeting a specific operation is then injected during each simulation. Mismatches can be easily detected when both faulty and golden simulations are performed. Each operation specific $IER_{insn}^{op_faulty}$ is obtained from 3000 simulations. The IER can also be derived analytically from Equation 5.1, where C_{op}^e and $M_{op}^{e_in, e_out}$ need to be obtained based on fault simulations. Here the experimental value is simply applied for higher estimation accuracy.

Table 5.1 shows $IER_{insn}^{op_faulty}$ s of instruction *alu_rrr* as an example. Table 5.1 also shows the application dependent weights of instructions for Sobel. The weights are used to calculate $p \cdot IER$, which constitutes one portion of the AER in Equation 5.2. Such weights can be obtained directly by the profiling tools of Processor Designer. Note that *alu_rrr_dc* and *alu_rrr_ex* operations are subdivided into several modes. This is because different modes of the same instruction type have distinct IER s and weights. The IER among different modes is the weighted average of IER s for all modes.

Table 5.1: Instruction-level reliability estimation

Operation	Mode	$IER_{insn}^{op_faulty}$	p_{sobel}^{insn}	$p \cdot IER$
fetch		0.512	0.148	0.0760
decode		0.623	0.148	0.0924
alu_rrr_dc	Total	0.199	0.148	0.0295
	add	0.268	0.081	0.0218
	sub	0.133	0.010	0.0013
	and	0.064	1e-4	6e-6
	or	0.111	0.056	0.0062
	xor	0.169	0.002	0.0003
alu_rrr_ex	Total	0.246	0.148	0.0547
	add	0.256	0.081	0.0208
	sub	0.249	0.010	0.0024
	and	0.109	1e-4	1e-5
	or	0.232	0.056	0.0130
	xor	0.215	0.002	0.0003
writeback_dst		0.853	0.148	0.1265

5.1.4.2 AER

When $IER_{insn}^{op_faulty}$ s for all operations and instructions are obtained from the testbenches, equation 5.2 is applied to estimate $AER_{op_faulty}^{app}$ based on the application profiling. Table 5.2 shows the estimation, experimental values and also relative deviation between both values averaged for three selected applications. In each experiment, one single bit fault with duration 1 clock cycle is injected randomly in time and location into the target operation. All analytical reliability estimation values can be obtained through *one single* simulation which consumes negligible amount of time while each experimental value comes from 10,000 LISA level fault simulation experiments, which requires around *five hours* each for Sobel and FFT and *12 hours* for IDCT. This is a significant improvement in the productivity and facilitates exploration by the application developer like the optimizations proposed in [162]. Naturally, for any change in the processor datapath or storage, the analytical model parameters need to be recomputed via benchmarking against instruction-set simulation-based or RTL-based reliability estimation flow.

Generally, for all three applications the estimated and experimental AER values of the same operation are close to each other. Regarding individual $AER_{op_faulty}^{app}$, *fetch*, *decode* and *writeback_dst* operations are apparently more vulnerable than the others since they reside on the paths of many operations. Besides, *address_generation* shows highest AER among all other operations, this happens since it is activated by *load* and *store* operations with direct access to the resources. *Nop* shows 0 error rates since it contributes nothing to the program execution. Compared among different applications, *ldc_ri_dc* is more vulnerable in FFT since coefficients are more frequently loaded in FFT than the others, while Sobel suffers more from faults in *alu_ri_dc* and *alu_ri_ex* since the compiler generates more assembly codes for calculation with immediate values.

For estimation accuracy, the results for operations with higher AER values show better matches. This happens since frequently called operations are more robust to the randomness during fault injection. Besides, AER s of operations which involve conditional behaviors such as *cmp_rr* and *bra* are highly dependent on the application characteristics, which makes it difficult to predict from IER s obtained using a standard testbench.

5.1.5 Summary

In this work, an analytical reliability estimation technique is presented, which facilitates fast reliability estimation for the target processor architecture with sufficient accuracy compared with instruction-set simulation-based estimation. The estimation accuracy of both the techniques are demonstrated through several embedded applications on a RISC processor and by benchmarking against an high-level fault injection.

Table 5.2: Reliability estimation for selected applications

Operation	$AER_{op_faulty}^{app}$						Rel.
	Sobel		FFT		IDCT		Dev.
	Est.	Exp.	Est.	Exp.	Est.	Exp.	
fetch	0.533	0.533	0.524	0.527	0.526	0.514	0.01
decode	0.629	0.635	0.595	0.606	0.610	0.616	0.01
writeback_dst	0.514	0.518	0.426	0.426	0.417	0.420	0.01
alu_rrr_dc	0.029	0.024	0.022	0.023	0.025	0.024	0.09
alu_rrr_ex	0.054	0.054	0.036	0.034	0.051	0.051	0.03
alu_rri_dc	0.041	0.043	0.020	0.018	0.018	0.018	0.05
alu_rri_ex	0.040	0.039	0.019	0.019	0.018	0.018	0.02
alu_rrri_dc	0.015	0.016	0.005	0.004	0.012	0.010	0.12
ld_rr_dc	0.074	0.070	0.056	0.055	0.083	0.078	0.05
address_gen	0.082	0.082	0.069	0.068	0.112	0.106	0.02
ld_mem	0.024	0.024	0.018	0.020	0.027	0.028	0.06
ldc_ri_dc	0.002	0.002	0.044	0.053	0.002	0.003	0.16
lui_ri_dc	0.003	0.005	0.004	0.005	0.003	0.004	0.35
st_rr_dc	0.026	0.025	0.024	0.024	0.042	0.041	0.02
st_mem	0.021	0.018	0.019	0.019	0.034	0.037	0.08
cmp_rr_dc	0.005	0.004	0.009	0.012	0.003	0.005	0.31
cmp_rr_ex	0.014	0.016	0.025	0.029	0.009	0.011	0.15
bra	0.025	0.018	0.041	0.035	0.031	0.028	0.17
branch_exe	0.011	0.011	0.021	0.021	0.011	0.015	0.11
branch_wb	0.012	0.012	0.014	0.014	0.015	0.015	4e-3
brau	0.023	0.023	0.024	0.023	0.022	0.024	0.04
brai	0.006	0.007	0.006	0.008	0.007	0.007	0.13
nop	0	0	0	0	0	0	0

5.2 Probabilistic Error Masking Matrix

In presence of reliability estimation techniques, design of reliable system is still a challenging problem. Designing a reliable processor requires thorough understanding of all the causes of failures such as external radiation, electromigration and thermal cycles. Furthermore, reliability brings forth trade-off with other design dimensions [3, 57, 89, 158]. Recent research treats reliability as a *cross-layer design issue* [45]. This stresses the fact that separate error mitigation techniques from individual design

abstractions may result in an over-protected system. The design should take the support of architectural and algorithmic error resilience [72,143]. However, this requires strong understanding of the fault propagation through different design abstractions, based on which resultant error properties such as location, timing and probabilities could be predicted. Such knowledge are difficult to acquire through analytical or fault injection techniques [35].

In particular approximate error prediction is an issue when *algorithmic reliability* is explored or when *inexact, probabilistic computing* [138] is performed. Similar research was pursued earlier for floating-to-fix point conversion in DSP design [73]. However, the error localities were restricted to variables (sizes of fixed points) and operators (saturation, rounding effects) without any architecture-level concern. Krishnaswamy et al. proposed a framework called Probabilistic Transfer Matrix (PTM) [160] which captures the probabilistic behavior of the circuit while estimates the approximate error probability of faults **inside** the circuit. Analytical study of error propagation could be potentially addressed using PTM. However, PTM suffers from scalability problem for large design due to its bit-level accuracy and is not initially designed for handling error masking effects. In [128] a statistical error tracking approach named RAVEN is introduced to analysis error effects across multiple design layers. The work is demonstrated on the IVM processor where the DUE (Detected Unrecoverable Error) and SDC (Silent Data Corruption) outcomes for soft error candidates are predicted. However, RAVEN analyses error propagation of large micro-architecture blocks such as a pipeline stage using averaged masking statistics, which implies increased amount of error due to various logic masking effects which depend on runtime processor behaviors.

Contribution In this work, a new algebraic representation named Probabilistic error Masking Matrix (PeMM) is first introduced to investigate the masking effects on errors occurring at the **inputs** of the circuits. Compared with PTM, PeMM has a reduced calculation complexity due to the scope of error focusing on coarse-grained signal level. Next PeMM algebra is integrated into a high-level processor design framework and represent logic error as an abstract data structure named *token*. The proposed approach is illustrated using *soft error* occurring at registers and memories, which are more susceptible to transient fault than combinational logic circuits [167]. An automated analysis flow is presented where the token propagation can be predicted using cycle-accurate instruction-set simulator while the error masking effects are carefully addressed using PeMM for individual micro-architecture unit. Fine-grained PeMM is also proposed which calculates nibble-wise or byte-wise error probabilities on data signals. Compared with RAVEN, several optimization techniques are introduced to increase the accuracy of prediction for runtime logic masking effects. Consequently, the significance of logic faults through design abstractions could be approximately predicted in earlier design phases.

5.2.1 Logic Masking in Digital Circuits

Faults within logic circuits are masked with certain probability before propagating to the circuits' outputs as visible errors. Such masking effects are seen due to several reasons as:

- Logic primitives performing algorithmic calculations have inherent ability of masking faults at inputs, which give error-free outputs.
- Micro-architecture features such as inter-stage data bypass can neglect the faulty input by replacing it with fault free input as a feedback from other pipeline stages.
- Faulty resources of processor such as registers and memory elements can never be read by computational circuit, giving always a fault free output.
- The faulty value of storage element or wires are overwritten before being read.

PTM [160] calculates the error probability of circuits' output by considering the logic circuits as a white box with faults inside logic blocks, which is the case in Figure 5.4a). The approach suffers from scalability problem for large circuits since the size of the PTM is $2^n \times 2^m$ where n and m imply the total number of bits for inputs and outputs. Besides, for large scale circuits the derivation of PTM can be extremely time consuming since PTMs of individual logic gates needs to be accumulated.

Figure 5.4: Faults in logic circuits

Probabilistic error Masking Matrix (PeMM) is proposed to address the scalability issue, where the faults reside in *inputs* of circuits only as in Figure 5.4b). In contrast to PTM, PeMM has the size of $m \times n$ for a circuit with n bits input and m bits output. The size of matrix can be further reduced depending on the level of error existence. For instance, n and m represent number of input and output signals when signal level error existence is considered.

5.2.1.1 PeMM Definition

Consider a circuit with n inputs and m outputs. The n inputs are labelled as $in_0, \dots, in_{n-1}$ and the m outputs as $out_0, \dots, out_{m-1}$. The PeMM P of the circuit is a matrix with dimension $m \times n$. Each entry in $P(out_i, in_j)$ represents error masking probability $M_{in_j}^{out_i}$,

Figure 5.5: Probabilistic error Masking Matrix (PeMM)

where $i \in [0, m - 1]$ and $j \in [0, n - 1]$. It shows the error masking effect on output out_i with regard to input in_j , where 0 means the error has been completely masked while 1 implies no masking effect at all. Note that $e_{out_i} \in [0, 1]$ so that value larger than 1 will be truncated. The inputs of the circuits are represented by a column vector I with dimension $n \times 1$. Entry in $I(j)$ represents the error probability e_{in_j} associated with input in_j . When the input vector is left multiplied with P , resultant output vector is represented by a column vector O with dimension $m \times 1$, whose entry shows the error probability e_{out_i} associated with output of circuits. Figure 5.5 visualizes the abstract circuit model with its PeMM.

Besides the entire circuit, PeMMs can also represent error masking effects of individual micro-architecture units. Such divide and conquer approach considers the circuit PeMM as the concatenation of PeMMs for architecture units. Figure 5.6 shows the architecture units for executing the ALU instructions and data signals between logic operations. The dimensions of selected PeMMs are shown based on the counts of input and output signals. In case that input faults are not completely masked, the unit outputs errors with certain probability.

5.2.2 PeMM for Processor Building Blocks

5.2.2.1 Combinational Logic Blocks

PeMM performs a transformation of error masking probability from logic inputs to outputs for linear circuits. However, such approach is not applicable for logic blocks with internal data dependencies. To address such issue, larger circuits are decomposed into logic sub-blocks with individual PeMMs. Figure 5.7 indicates PeMM decomposition based on data dependencies, where the large logic block *alu_ex* is split into 3 sub-blocks. Signals *alu_in1* and *alu_in2* connect first two sub-blocks while *alu_out* connects the last two sub-blocks. PeMMs for sub-blocks are characterized individually. The intra-token pool is used to keep the temporary tokens with error probabilities for processing by following logic sub-blocks.

Figure 5.6: Logic blocks involved for ALU instruction

5.2.2.2 Control Flow inside Logic Block

The other factor contributes to the inaccuracy of PeMM estimation is the dynamic control flow within logic blocks. The run-time circuits masking capability shows significant difference compared with purely random characterization. This can be seen from the behavioral description of circuits shown in Figure 5.8, where a 3-to-1 multiplexer is generated during logic synthesis for the if/else statements. During execution, various active path shows different PeMM for the same logic block which leads to exclusive elements in PeMM. Random characterization results in a PeMM elements of [0.33 0.33 0.33] indicating error probability on each path is statically masked to 33%. To increase accuracy, additional *helper_signal* is adopted to indicate selected branch dynamically and fill the corresponding elements in PeMM. In Figure 5.8, vector [1 0 0] is filled into PeMM when the first branch of if statement is selected. Such approach reviews trade-off between accuracy of characterization and modelling efforts.

5.2.2.3 Sequential Logic and Memory

Other than combinational logic, sequential logic and memory exhibit no internal error masking effects on their inputs. Such elements have equal number of inputs and outputs whose PeMM can be modelled using identity Matrix $I_{m \times m}$, where m is the number of inputs and outputs. For pipeline registers, errors on input ports are mapped consistently to corresponding output ports during pipeline shift. For RegisterFile, input errors are stored for write access while same errors are loaded during read access.

Figure 5.7: Decomposition of large logic block using PeMM

Similarly, PeMM for memory is modelled as identity matrix with m equalling to the number of storage cells inside the memory. PeMM does not model error occurring inside sequential logic, which can be alternatively addressed using PTM [160].

5.2.2.4 Inputs with Multiple Faults

Multiple faults on the inputs of the circuit also affects PeMM characteristics. Matrix multiplication sums up the contribution of each input error after individual masking effects, which achieves good masking accuracy for non-algorithmic operations. For algorithmic operations multiple input faults can strongly vary the values of $M_{in_j}^{out_i}$ compared with fault on single input, especially in the case of correlated faults which are completely or partially originated from the same fault. Correlated faults are possible to inversely affect each other and even cancel the resultant errors depending on the algorithmic operation, such as in the case of XOR operator with bit-flip faults on both inputs.

Figure 5.8: Control flow handling for PeMM

5.2.3 PeMM Characterization

PeMM elements are characterized through high-level behavioral simulation where primary inputs of logic blocks are subject to fault injection. For a specific circuit with random data inputs, the masking probability $M_{in_j}^{out_i}$ can be acquired by averaging the error probability on out_i among multiple experiments, where single bit-flip fault is injected onto input in_j in each experiment.

5.2.3.1 Accuracy of PeMM Characterization

The characterization testbench compares the fault-free (golden) simulation results with fault injection on the specific inputs. To characterize each element in PeMM with a required confidence level and confidence interval, the number of experiments is determined by all the possibilities of bit flips in given input space (population) according to [40]. For a circuit with n inputs of m bits each, the size of input space is $2^{m \times n}$, the overall size of possible experiments equals $2^{m \times n} \times m$ with the possibilities of random bit-flip in a given input space. For instance, a circuit with 2 inputs, each 32

bits wide, requires 9604 experiments to achieve 95% confidence level with confidence interval of 1.

5.2.3.2 Fine-grained PeMM

As stated previously, granularity of PeMM varies depending on the level of error existence. When the input vectors represent signal-level error probabilities, non-zero values in output vectors indicate the existence of errors in particular signals. An instinctive extension to the approach would be the focus of error existence in smaller granularity such as byte or nibble level, where PeMM predicts not only signal-level error existence but also in which byte/nibble the error exists. This could be of particular importance in the field of approximate application, where the achieved QoS could be traded off with other design metrics such as power consumption and area overhead.

Fine-grained PeMM can be created using additional look-up-table for values of $M_{in_j}^{out_i}$ as in Table 5.3, where byte-level masking probabilities for selected algorithmic operations are listed. The first column represents the targeted operations while the second column forms a *Key* variable showing in which bytes the faults locate for both inputs of logic primitives. For instance, key 13 shows faults in 1st byte of first input and 3rd byte of second input while key 10 shows no fault in second input but only 1st byte of the first input. The byte-wise $M_{in_j}^{out_i}$ shows the probabilities of error existence in particular output bytes. Depending on targeted field of application, granularity can be further fine-grained, which requires additional efforts for characterization. Such as single input fault in 1st byte of *SUB* operation can result in errors in 2nd or even 3rd bytes with reduced probability, whereas for *AND* operation no cross bytes error could be resulted from single input fault. When faults exist in multiple bytes of the same input, expected masking probabilities could be interpolated based on byte-level error probabilities.

<i>Operation</i>	<i>Key</i>	<i>Byte-wise $M_{in_j}^{out_i}$</i>				
SUB	10	1.000000	0.126830	0.000520	0.000000	
OR	22	0.000000	0.721690	0.000000	0.000000	
AND	10	0.499030	0.000000	0.000000	0.000000	
AND	13	0.500400	0.000000	0.499900	0.000000	
XOR	33	0.000000	0.000000	0.873990	0.000000	

Table 5.3: Examples of PeMM elements with byte-level granularity

Figure 5.9 and 5.10 shows the examples of byte-level and nibble-level PeMM. Each single element in word level PeMM is expanded as 4×4 sub-matrix in byte-level PeMM and 8×8 sub-matrix in nibble-level PeMM. The indexing label *out/in* represents the sub-matrix with regard to the input signal *in* and output signal *out*. The

Level_1 -----					Level_2 -----				
alu_in2/mode					alu_out/opcode				
0.228	0	0	0		0.98	0	0	0	
0.201	0	0	0		0.981	0	0	0	
0.196	0	0	0		0.978	0	0	0	
0.232	0	0	0		0.979	0	0	0	
alu_in2/shifter_in1					alu_out/alu_in1				
0.355	0.122	0.123	0.109		0.819	0	0	0	
0.07	0.361	0.124	0.131		0.047	0.796	0	0	
0.067	0.069	0.368	0.156		0	0.059	0.794	0	
0.066	0.051	0.055	0.373		0	0	0.043	0.769	
alu_in2/shifter_in2					alu_out/alu_in2				
0.358	0	0	0		0.792	0	0	0	
0.334	0	0	0		0.043	0.786	0	0	
0.282	0	0	0		0	0.049	0.79	0	
0.208	0	0	0		0	0	0.048	0.794	
Level_3 -----					Level_3 -----				
EX_MEM_WBV/dst					EX_MEM_WBV/alu_out				
0.058	0	0	0		0.919	0	0	0	
0.058	0	0	0		0	0.943	0	0	
0.057	0	0	0		0	0	0.942	0	
0.058	0	0	0		0	0	0	0.935	
EX_MEM_BPR/dst					EX_MEM_BPR/alu_out				
0	0	0	0		0	0	0	0	
0	0	0	0		0	0	0	0	
0	0	0	0		0	0	0	0	
0	0	0	0		0	0	0	0	

Figure 5.9: Byte-level PeMM

overall error probabilities on a specific segment of signal *out* is the sum of contribution from propagated error through all sub-matrix which has the same output signal and segment. Take the element *alu_out/alu_in1* for instance, it is observed that the expansion of error into neighbour segments with reduced error probabilities once upon fault is injected in a single segment. Further more, nibble-level PeMM shows the cross-section error propagation more clearly since mismatches on finer segments are characterized.

5.2.4 Approximate Error Prediction Framework

The approximate error prediction framework is proposed using PeMMs for individual logic blocks. In this work it is integrated with LISA-based processor design flow [182] while the approach is generic for any architecture simulator such as Verilog and SystemC simulators. Figure 5.11 shows an overview of the framework.

The flow consists of the preparatory and execution stage. In preparatory stage, cycle accurate instruction-set simulator (ISS) running specific applications is first generated from processor description using ADL LISA [2]. With the simulator extension, ISS is prone to fault injection where user can configure fault information using graph-

Level_2 -----								
alu_out/opcode								
0.869	0.898	0	0	0	0	0	0	0
0.898	0.894	0	0	0	0	0	0	0
0.899	0.899	0	0	0	0	0	0	0
0.88	0.904	0	0	0	0	0	0	0
0.885	0.893	0	0	0	0	0	0	0
0.893	0.908	0	0	0	0	0	0	0
0.904	0.895	0	0	0	0	0	0	0
0.88	0.867	0	0	0	0	0	0	0
alu_out/alu_in1								
0.79	0	0	0	0	0	0	0	0
0.088	0.799	0	0	0	0	0	0	0
0.007	0.094	0.788	0	0	0	0	0	0
0	0.006	0.089	0.796	0	0	0	0	0
0	0	0.003	0.087	0.801	0	0	0	0
0	0	0	0.004	0.104	0.817	0	0	0
0	0	0	0.001	0.012	0.079	0.783	0	0
0	0	0	0	0.002	0.003	0.087	0.79	0
alu_out/alu_in2								
0.799	0	0	0	0	0	0	0	0
0.106	0.811	0	0	0	0	0	0	0
0.008	0.088	0.776	0	0	0	0	0	0
0.002	0.005	0.105	0.807	0	0	0	0	0
0	0	0.007	0.099	0.795	0	0	0	0
0	0	0	0.009	0.088	0.803	0	0	0
0	0	0	0	0.006	0.092	0.81	0	0
0	0	0	0	0	0.009	0.088	0.792	0

Figure 5.10: Nibble-level PeMM

ical interfaces. An extra LISA code parser is used to translate LISA description into abstract circuit models, which contain the information on the directed acyclic graph (DAG) of LISA operations and the inputs and outputs resources for individual architecture units. The PeMM characterization module translates the behavior of processor architecture units into C-based functions with signal inputs and output as function arguments. Testbenches are generated to inject random faults to function inputs, where PeMMs are fast characterized. Special language directives are checked by the LISA parser for finer PeMM characterization according to techniques in section , 5.2.2.2 and 5.2.3.2.

The execution stage starts with the creation of token by user using a graphical interface. The token represents the fault in terms of error probability, along with other fields representing the micro-architectural and timing information required to track the token as it propagates. The cycle accurate error tracker works for generic processor models, which tracks the token propagation with possible masking effects applying PeMMs for architecture units. The reports on predicted errors are generated by the error tracker, which documents detailed paths of token propagation and various masking effects through architecture units.

Figure 5.11: Error tracking and prediction framework

5.2.4.1 Error Representation

In contrast to fault injection method where the value of resources is changed dynamically, token is created as an abstract data structures which does not change the resource values. Instead, error probability is updated during token propagation. Error probability is initially set to 1 during token creation. When error probability is masked to 0 the token is removed. Hardware resource ID and memory array index are associated with each token so that the PeMM addresses the corresponding token correctly. Specific resources are possible to contain multiple *sub-tokens*, for instance the *instruction* register, which consists of multiple decoding fields such as *opcode*, *source* and *destination* operands.

5.2.4.2 Token Tracking

Since no actual faults are injected but only abstract tokens, the simulator maintains correct execution flow while indicates potential errors. Algorithm 1 shows the token tracker routine called between consecutive processor control steps. The routine starts with the activation analysis of LISA operations. If any operations whose inputs contain tokens are activated, the tokens are updated by PeMMs and propagate to the outputs of the operation by the end of the cycle. Due to synchronized register be-

Algorithm 1 Token tracking routine

```

1: function TRACKTOKEN(*op, *token, *PeMM)
2:   for all op_id do                                     ▷ Create tokens by activation analysis
3:     if op[op_id] is active then
4:       if  $\exists$  token[tk_id] in op[op_id].inputs then
5:         Update with PeMM[op_id] for op[op_id];
6:         Schedule to create tokens in op[op_id].outputs;
7:         New tokens labelled as high priority;
8:       end if
9:     end if
10:  end for
11:  for all tk_id do                                       ▷ Create tokens by pipeline behaviors
12:    if token[tk_id] is in pipeline registers then
13:      Schedule to remove token[tk_id];
14:      if token[tk_id] is not in last pipeline stage then
15:        Schedule forwarding token in next stage as low priority;
16:      end if
17:    end if
18:  end for
19:  Remove_tokens();
20:  Create_tokens();                                       ▷ Create/remove tokens at end of the control step
21: end function

22: function CREATE_TOKENS
23:   for all tokens in schedule creating list do
24:     if  $\exists$  old token in new location then
25:       Overwrite old token;
26:     end if                                             ▷ Overwrite existing tokens
27:     if multiple tokens are scheduled in the same location then
28:       Create token with high priority;
29:     else                                             ▷ Forwarded tokens have less priority
30:       Create token;
31:     end if
32:   end for
33: end function

```

haviors, the tokens cannot be immediately created or removed before the completion of analysis for the current cycle, but are scheduled for creation and removal. After activation analysis for operations, the tokens in pipeline registers are forwarded to the next pipeline stage. However, forwarded tokens have less priority compared with the ones created from the active operations. For memory and register files, old tokens are overwritten by new arriving ones.

5.2.5 Results in Error Prediction

To demonstrate the usability of the approximate error prediction flow, several case studies using an embedded RISC processor from Synopsys Processor Designer [182] are presented. The processor has five pipeline stages with fully bypassing and forwarding functionality. Verilog models are generated automatically for fault injection experiments.

5.2.5.1 Accuracy and Speed-up

The predicted error probability is compared with Verilog-based fault injection experiments [42], where the faults can be injected into physical resources such as pipeline registers, RTL signals, register file and memory arrays of the processor in Verilog representation.

Accuracy Comparison among PeMM Configurations In the first experiment, a simple testbench containing all types of instructions for the RISC processor is designed, which processes data in a loop using general purpose register arrays and stores the final data into memory. Error prediction results using different modes of PeMM construction are compared with RTL fault injection. In each fault injection experiment single bit-flip fault is randomly injected among 32 bits of register containing input data of the testbench. 1,000 such fault injection experiments with random input values are performed to calculate the average error probabilities on selected hardware resources. On the contrary, proposed error prediction analysis is performed only once to show the predicted error probabilities with the same fault configuration.

Figure 5.12 shows the prediction results using different PeMM configurations on selected hardware resources, which include general purpose registers R[1] to R[15] and program output value to data memory. It is indicated that the initial PeMM without matrix decomposition achieves least similarity compared with fault injection, while proper PeMM decomposition and appliance of assistant signals for control flow prediction shorten the gap significantly. When assistant signals are inserted for control paths in all logic blocks, the predicted error probabilities and locations perfectly match the results from fault injection.

Error Prediction for Embedded Applications Several embedded benchmarks are used for demonstrating the accuracy and timing of error prediction framework against RTL fault injection. For both groups of experiments, one token/bit-flip fault is created/injected in the same resource location at same time instances. For benchmarking purpose, the hardware locations which result in errors in register file or data memories are targeted on purpose, instead of being masked during propagation. Table 5.4 shows the word level error probability on selected hardware registers and data memory locations upon completion of applications. The advance mode of PeMM using both helper signal and decomposition is adopted for prediction.

It is noted that the error probabilities from fault injection experiments approaches the predicted values as number of experiments grows. The exhaustive data input patterns and random faults generation during PeMM characterization phase contribute to the prediction accuracy. Table 5.4 also shows the speed comparison between prediction and RTL simulation with 2,000 experiments. Token tracker takes only several seconds against scale of hours consumed by fault injection experiments. On average, token tracking achieves 25,000x speed-up compared with RTL simulation with 2,000 experiments.

Figure 5.12: Comparison of error prediction accuracy for different modes of PeMM against fault injection [42]

Apps	Traced Resource	Array Index	Token tracker		Verilog fault injection [42]			
			Error Prob	Time (sec) one token	Error Prob			Time (hours)
					Number of experiments			
					1000	2000	3000	2000
Cordic	R	12	0.75	2.5	0.89	0.81	0.79	17.31
CRC	dmem	0x100017	0.03	3.4	0.07	0.04	0.03	23.43
IDCT	dmem	0x100034	0.98	2.9	1.00	1.00	0.99	19.41
Rijndael	dmem	0x10002F	0.48	3.9	0.61	0.66	0.59	28.24
Sobel	R	7	0.24	2.4	0.30	0.28	0.29	16.63
Viterbi	dmem	0x100078	0.90	4.4	0.98	0.95	0.93	33.30

Table 5.4: Speed and accuracy using proposed framework

5.2.5.2 Timing Overhead for Token Tracking

Overhead of Preparatory Stage The preparatory stage consists of two phases, the parsing stage analyses processor model and converted into C-based characterization testbenches, while the characterization phase generates PeMM elements based on the pre-defined PeMM modes. The parsing phase splits larger logic blocks and inserts

helper signals based on user defined language constructs to decomposed larger logic blocks.

The preparatory stage analyses 42 operations for the RISC processor. Table 5.5 shows the timing overhead for the preparatory stage on the host machine with Intel Core i7 CPU at 2.8 GHz. In this experiment, 100,000 characterizations are performed for each element in PeMM. It is observed that analysis of advance PeMM modes consumes more time in both phases.

	Initial PeMM (sec)	Split only PeMM (sec)	Split+full ass-signals PeMM (sec)
Parsing	0.14	0.17	0.26
Characterization	2.80	2.85	4.75

Table 5.5: Processing time for automated PeMM preparation

Timing Analysis against Token Counts The token tracking framework is an extension of the cycle accurate instruction-set simulator from Synopsys Processor Designer [182]. Table 5.6 shows the timing overhead caused by the token tracking against original simulation using several embedded benchmarks. Analysis without any token adds in average 28.4% simulation overhead due to the additional analysis for operation activation in each clock cycle. When there is one token created, the tracking routine consumes 6.7% overhead. When 20 tokens are created, in average 79.3% overhead is added. The exact overheads differs among experiments due to the host machine usage and randomness in both injected location and time instance, where in most of the cases tokens are masked during execution. Tracking of multiple tokens scales the simulation time in a linear fashion due to the fact that tokens are managed in the unordered hash map with timing complexity of $O(1)$ for element searching [38].

Apps	Original simulator [182]	Token tracker					
		0 token		1 token		20 tokens	
	No fault (sec)	(sec)	+%	(sec)	+%	(sec)	+%
Cordic	1.7	2.3	35	2.5	9	3.3	32
CRC	2.1	3.2	52	3.4	6	6.2	82
IDCT	2.5	2.8	12	2.9	4	7.0	141
Rijndael	2.3	3.5	52	3.9	11	5.8	49
Sobel	1.8	2.1	17	2.4	14	6.4	167
Viterbi	3.3	4.3	30	4.4	2	8.1	84
Average	-	-	28.4	-	6.7	-	79.3

Table 5.6: Timing overhead analysis against architecture simulator

Timing Analysis among Modes of PeMM Figure 5.13 shows time consumed by several embedded benchmarks using different modes of PeMM. It is observed that run-time efforts of error prediction using split PeMM with helper signals increase, which shows a trade-off between prediction accuracy and analysis time.

Figure 5.13: Run-time among different PeMM modes

5.2.5.3 Prediction of Application-level Error Locations

The advantage of error prediction is far more than its speed and accuracy. Fault injection is impossible to track where exactly the errors result in the huge memory address space. The ability to predict error locations especially in memory array assists the designer to find the potential error effects in application-level. The feature of data prediction is demonstrated using the median filter [83] where both input and output images are shown in Figure 5.14. When two tokens are injected in the memory location which contains the input image, the affective regions are predicted in the output image accordingly. Such prediction matches the algorithm specification in [83], where the value of each pixel in the output image is related to the median value of the pixel at the same position and its surrounding 8 values in the input image.

5.2.6 Summary

In this work, probabilistic error masking matrix is introduced to investigate the error masking effects of logic circuits. A fast and approximate error prediction framework is introduced which tracks the paths of error propagation and estimates error probabilities. Vulnerability of hardware resources can be easily estimated, while location and significance of errors are predicted. Benchmarking with state-of-the-art RTL fault injection indicates that the proposed framework achieves high accuracy of error prediction and significant speed-up.

Figure 5.14: Error prediction for median filter application

5.3 Reliability Estimation using Design Diversity

Many reliability enhancement techniques across design abstractions have been proposed [100], among which redundancy is widely applied for increasing the *data integrity* which indicates the probability that the system either produces correct output or detects incorrect outputs. Redundancy takes advantage of repeated execution of same program on multiple hardware copies to verify the correctness of the results. In simultaneous multi-threaded processor, redundant Multi-Threading (RMT) [154] are exploited to take advantage of the idle hardware threaded. In [134] the author extended RMT approach to multiprocessor architectures, which is known as Chip-level Redundant Threading (CRT), where a leading thread in one core is verified by a trailing thread in another core. Besides hardware redundancy techniques, software based redundant execution takes advantage of the idle instruction slots for on-line verification [152]. Large performance overhead is expected for the shadow program to fill the available instructions slots.

The key idea of a redundant system is *duplication*, where two modules are used to verify the correctness of the outputs based on a comparator. However, the duplex system suffers from common-mode failures (CMFs) when a single type of failure affects more than one module at the same time [115].

Strong need is required to *quantify* the effects against CMFs of a redundant system. Several reliability estimation techniques have been proposed to investigate the behavior of circuits under faults, which utilize either fault injection [88] or analytical prediction [23]. However, neither technique provides a quantifiable metric for a redundant system. The problem has been addressed for circuit-level design using *design diversity*, which was initially proposed in [13] to protect the duplex system against CMFs and has been applied in [104] [153] for design of robust systems. Mitra et al. [130] further develops design diversity as a quantifiable metric for evaluation of duplicated system.

Figure 5.15: Duplex and multiplex redundant systems

Simulation-based reliability estimation methods fail to generate confidence regarding the accuracy [35]. Existing circuit-level reliability techniques do not scale to architectural abstraction and on the other hand, due to growing number of diverse architectural blocks in modern SoC, there is a strong need to understand the architectural reliability metrics combined with circuit-level models.

Contribution In this work, design diversity is utilized as a metric for architecture-level reliability evaluation which is validated on diverse processing architectures. A novel graph-based analysis is proposed which jointly quantifies circuit-level design diversity and architecture-level operator exclusiveness. The proposed techniques are useful to evaluate the reliability of diverse classes of architectures. Case studies on several embedded computing architectures are demonstrated through the analysis on architecture and application-level design diversity, as well as estimation of the system Mean-Time-to-Failure.

5.3.1 Design Diversity

In a duplex system as shown in left side of Figure 5.15, two modules are used to verify the correctness of the outputs which may suffer from CMFs. Design diversity indicates that a duplex system with different implementations of its modules will probably result in different error outputs facing CMFs, which is detectable by comparing the outputs. For a system consisting of more than two replicated modules as in a duplex system, it is named as a multiplex system as shown in right side of Figure 5.15.

Diversity $d_{i,j}$ with respect to fault pair (f_i, f_j) is defined in Equation 5.3. The factor $k_{i,j}$ represents the *joint detectability*, which is the number of input combinations that produce identical erroneous output in the duplex system. n is the number of input bits, thus 2^n is the amount of all input combinations.

$$d_{i,j} = 1 - \frac{k_{i,j}}{2^n} \quad (5.3)$$

For a given faulty duplex system, the design diversity metric is defined to be the expected value of the diversity with respect to all the possible fault pairs, which can be shown in Equation 5.4. In the formula, $d_{i,j}$ is the diversity with respect to fault pair (f_i, f_j) . $P(f_i, f_j)$ is the probability that fault pair (f_i, f_j) happens. $\sum_{(f_i, f_j)}$ collects all the possible fault pairs in the duplex system. Thus the system diversity D represents the probability that a duplex system produces either correct outputs or detectable errors when two modules are affected by faults. Consequently, system error probability E shows the probability that a duplex system generates undetectable erroneous output as shown in Equation 5.5.

$$D = \sum_{(f_i, f_j)} P(f_i, f_j) d_{i,j} \quad (5.4)$$

$$E = 1 - D \quad (5.5)$$

The design diversity metric is extended as in Equation 5.6 for a multiplex system, where $d_{i,j,...,k}$ is the diversity with respect to fault set $(f_i, f_j, ..., f_k)$. The design diversity of a multiplex system represents the probability that a multiplex system produces either correct outputs or detectable errors when all its modules are affected by faults.

$$D = \sum_{(f_i, f_j, ..., f_k)} P(f_i, f_j, ..., f_k) d_{i,j,...,k} \quad (5.6)$$

Design diversity for duplex system can be calculated through exhaustive simulation. Efficient diversity estimation techniques are proposed in [131]. [130] also performs reliability analysis using diversity metric with extensive case studies on circuit-level duplex system, which shows that diversely implemented duplex pair is significantly reliable against the one containing the same implementation.

Figure 5.16 shows 2 types of implementation of 1 bit full adder and full subtracter, while the design diversity metric calculated using Equation 5.4 under worst case condition [130] is listed in Table 5.7. The calculation is based on exhaustive simulation under fault injection on all possible fault pairs. For both circuits, the duplex system with different types of implementation achieves better design diversity, which indicates the high reliability against faults.

Figure 5.16: Implementation for Full Adder (FA) and Full Subtractor (FS)

Logic function vs Duplex type	T1 + T1	T2 + T2	T1 + T2
Full adder	0.4637	0.3608	0.6026
Full subtractor	0.4028	0.2504	0.5238

Table 5.7: Design diversity for duplex pairs in Figure 5.16

5.3.2 Graph-based Diversity Analysis

In previous work, design diversity is applied for circuits level reliability analysis. In this work it is extended into architecture-level analysis. Reliability analysis on major SoC building blocks such as RISC and VLIW processors, as well as reconfigurable architectures like CGRA, are presented. A two phase analysis flow is proposed as following:

1. For each type of operator, quantify number of *conflict* functional units which can be executed *simultaneously* using architectural exclusiveness analysis as shown in this section.
2. Quantify circuit-level diversity for the conflict functional units with regard to each type of operator using technique in section 5.3.1.

The proposed analysis method indicates the *maximal* design diversity that the architecture can achieve, which jointly considers duplication techniques in architecture

Figure 5.17: Directed acyclic graph with ISA coding for ADL model

and circuit-level. To facilitate the proposed analysis, the estimation flow is designed in a high-level architecture design framework using ADL LISA. In this section, ADL LISA is first discussed from graphical perspective, followed by the graph-based representation and diversity analysis.

5.3.2.1 Graph Representation in LISA Language

LISA 2.0 [2] has been used to describe various architecture variants. An arbitrary LISA model is represented as a Directed Acyclic Graph (DAG) $D = \langle V, E \rangle$. V indicates functional unit where E represents activation of the child units from the parent ones. Figure 5.17 shows the DAG for a 5 pipeline stage RISC processor model, which consists of 4 groups of instructions e.g. Arith, Logic, Memory and Control. The instruction encoding for specific operators is shown in gray boxes, which is described as different coding fields. Each coding field is either a terminal field containing bits of '0' or '1', or a non-terminal field referring to the coding of child operators.

5.3.2.2 Exclusiveness Analysis

Exclusiveness of operators represents whether the operators can be executed in the same clock cycle. The analysis was performed in [206] for resource sharing of mutually exclusive operators. The exclusiveness information for LISA-based architecture descriptions can be extracted from the instruction encoding fields and activation edges in DAG, which is represented in a *conflict graph* as shown in Figure 5.18. In the conflict

Figure 5.18: Conflict graph for selected operators in Figure 5.17

graph, an edge between 2 function blocks indicates that they are *not* mutually exclusive, which is also called *conflict*. Operators from different pipeline stages, which are represented in different colors, are always conflict and for simplicity, edges between those are *not* shown. In Figure 5.18, operator Arith is conflict with Decode, Add, Sub, And and Or but exclusive with the rest operators.

5.3.2.3 Diversity Analysis

A duplex redundant system requires the *simultaneous* execution of *same* logic function on duplicated hardware copies. The information on simultaneous execution can be extracted from the conflict graph using exclusiveness analysis, while the information on functions of logic blocks are directly acquired from LISA DAG. A novel graph representation, named as *Conflict Multiplex Graph* (CMG), is proposed for illustrating following information:

Theorem 5.3.1 *CMG represents exclusiveness by colors, where the operators with same color are mutually exclusive.*

Theorem 5.3.2 *CMG represents functionality by edges, where solid edge between operators indicates identical duplex and dashed edge shows diverse duplex.*

An example of CMG with the corresponding DAG for RISC processor is shown in Figure 5.19. For simplicity, only CMG for EX pipeline stage is presented which contains 7 operators. Compared with Figure 5.17, decode unit activates 2 additional operators based on the coding field *Chk*, which is applied for checking Arith and Logic operators. Since *All_insn* and *Chk* originate from different coding fields in ISA, the checking operators are conflicting with Arith and Logic operators, which are represented in different colors. By carefully providing the input operands, Mac operator is capable to perform the functionality of Add, Mul and Sub with a diverse logic

Figure 5.19: Directed acyclic graph and conflict multiplex graph

implementation so that they are connected by dash edges. And1 and And2 operators are identical implementation and conflict with each other, which is connected by solid edge. No further edges exist in CMG since the rest pairs of operators are either mutually exclusive or not able to perform the same functionality.

To quantify architecture-level diversity, the amount of duplex pairs for specific functions are identified using CMG, while the design diversity for each pair is calculated based on the technique in Section 5.3.1. Table 5.8 summarizes the performed logic functions and implementation of duplex pairs from CMG in Figure 5.19. The diversity value for each pair is calculated using exhaustive simulation.

Logic function	Implementation	Design diversity
Add	Op_Add + Op_Mac	0.7243
Sub	Op_Sub + Op_Mac	0.7481
Mul	Op_Mul + Op_Mac	0.6160
And	Op_And1 + Op_And2	0.4287

Table 5.8: Duplex pairs for EX pipeline stage in Figure 5.19

Figure 5.20: Conflict multiplex graph for TMR Architecture

5.3.2.4 CMG for Several Architecture Variants

In this section CMGs for several architecture variants are presented while the corresponding duplex systems are identified. Note that the analysis of CMG-based diversity quantification identifies the *potential* of functional duplication for different architectures. However, huge engineering efforts are required to fully use such duplications in the run-time, which mainly result from the techniques during software design and/or compiler optimization flow. The detailed optimization techniques will be part of the future work.

TMR Triple Modular Redundancy uses three copies of logic units for verification of the protected function. Figure 5.20 shows an example of TMR scheme applying to Add and Sub operators. Other than a duplex system, TMR creates a multiplex system containing three implementations, which can be either identical or diverse. For instance, Add2 and Add3 are identical while Add1 differs from them as a result of different gate-level implementation. The design diversity for multiplex system is calculated using Equation 5.6. The multiplex system consists of three modules for TMR scheme. Note that even though Add2 and Add3 are conflict with all blue operators, the micro-architecture limit them from forming duplex pairs with operators other than Add1.

URISC URISC is proposed in [148] which adopts the Turing complete instruction *subleq* in the co-processor for diverse duplication of logic instructions in the MIPS processor core. Figure 5.21 presents the CMG graph for URISC architecture where the *subleq* operator in co-processor forms duplex pairs with conflict operators in the MIPS core. Taking advantage of Turing completeness, *subleq* makes pairs of all EX stage operators in the MIPS core.

Figure 5.21: Conflict multiplex graph for URISC Architecture

	alu	nop	cmp	mem	branch
syllable0	✓	✓	✓	✓	✓
syllable1	✓	✓	✓	✓	
syllable2	✓	✓			
syllable3	✓	✓			

Figure 5.22: Conflict multiplex graph for VLIW Architecture

VLIW In VLIW processor, multiple instruction syllables are used for parallel execution. Figure 5.22 shows the CMG for VLIW architecture with four syllables. The operators activated in each instruction syllable are conflict with the operators from the other ones so that multiplex system can be formed. For instance, Add1 can form identical duplex pair with Add2, Add3 and Add4, while also non-identical pair with Sub2, Sub3 and Sub4.

CGRA Coarse-Grained Reconfigurable Architecture consists of large number of processing tiles interconnected by a specific network topology. Each processing tile contains several pre-fabricated functional units (FUs) which are reconfigured by the synthesis tool. The run-time functions in each tile e.g. Add or Sub, are selected by hardware multiplexer using configuration bits, so that they are mutually exclusive. On the other hand, FU in one tile is conflict with FUs from other tiles since each tile has freedom in decision of its functions. Figure 5.23 shows the CMG for a CGRA architecture with six tiles, where multiplex system with huge amounts of implementations is formed for each logic function.

Figure 5.23: Conflict multiplex graph for CGRA Architecture

5.3.3 Results in Diversity Estimation

In this section the case studies of diversity estimation for several embedded architectures are first presented. In the next application-level design diversity is investigated based on instruction statistics. Furthermore, architectural design diversity is utilized to estimate the system MTTF metric.

5.3.3.1 Architecture Diversity Evaluation

The evaluation results are presented for three architectures including RISC, VLIW and CGRA, which are listed in Table 5.9. Four operators implemented on all architectures are selected as evaluation targets, which are Add, Sub, Sll, Srl. For each architecture, two variants are chosen which contain either identical or diverse modules of the selected operators. In case of diverse system, equal number of both module types is considered. Equation 5.6 is used for diversity evaluation on the multiplex systems.

Figure 5.24 shows the estimated architecture diversity values. Diversity of all operators shows similar trend among architecture variants. It is obvious that more modules in a multiplex system lead to higher diversity value. With the same number of modules, systems with diverse implementation achieve higher diversity values than the identical ones. Note that the RISC architecture with two diverse modules is comparable with VLIW with four identical modules with regard to diversity.

	No. modules type 1 / operator	No. modules type 2 / operator
RISC - Identical	2	0
RISC - Diverse	1	1
VLIW - Identical	4	0
VLIW - Diverse	2	2
CGRA - Identical	6	0
CGRA - Diverse	3	3

Table 5.9: Architecture variants of design diversity evaluation

Figure 5.24: Design diversity of architecture variants

5.3.3.2 Application-level Diversity Evaluation

The architectural design diversity of logic operators also provides a design metric for application analysis. Application-level design diversity is proposed in Equation 5.7, where $P_{op,app}$ is the probability of operator op among all operators in application app . $\sum_{op}$ collects contribution of diversity from all targeted operators. To achieve a high application-level design diversity thus a low error probability according to Equation 5.5, it is essential to match the operator op with high percentage in an application to a multiplex system with high architecture diversity with regard to op .

Figure 5.25: Application-level design diversity for PD_RISC processor

$$D_{app} = \sum_{op} P_{op,app} D_{op} \quad (5.7)$$

Application-level design diversity is evaluated using several embedded benchmarks on the PD_RISC processor from the IPs of Synopsys Processor Designer [182]. CoSy [186] based C compiler is used for building the benchmarks. The statistics on instruction profiling are acquired from the generated cycle-accurate instruction-set simulator. Diversity values are evaluated using Equation 5.7.

Figure 5.25 shows the application-level diversity evaluation for PD_RISC processor. Both diverse and identical implemented duplex systems are adopted, while Add, Sub, Srl, Srl are selected as targeted operators. For different applications, diverse system shows better diversity than identical ones. The absolute values differ among applications due to the difference in probabilities of the operators.

5.3.3.3 Mean-Time-To-Failure Estimation

$MTTF_{op}^{arch}$ for a specific operator op of the architecture $arch$ can be estimated using the failure rate λ_{op}^{arch} by Equation 5.8. Assuming a transient bit-flip fault model, using Equation 5.9, λ_{op}^{arch} is further derived from $P_{op}^{1fault,arch}$, which is the probability of 1 fault injected in *all* modules of multiplex system with operator op in architecture $arch$, and operator error probability E_{op}^{arch} , which equals $1 - D_{op}^{arch}$ as in Equation 5.5 and D_{op}^{arch} is the diversity of multiplex system with operator op in architecture $arch$. In Equation 5.10, $P_{op}^{1fault,arch}$ is further related to the architecture dependent product of

module-level fault probability $P_{op,i}^{1fault}$. It corresponds to the division of area estimation of the operator $A_{op,i}$ by the constant $A_{1fault/hour}$, which is the size of area that 1 fault happens per hour under a specific environmental condition. Such condition can be estimated by the reciprocal of Failure-in-Time (FIT) [69] in Equation 5.11. In this work, The FIT is assumed for instance as $10^{-4}cph/\mu m^2$, where the unit stands for fault count per hour (cph) per unit area (μm^2).

$$MTTF_{op}^{arch} = \frac{1}{\lambda_{op}^{arch}} \quad (5.8)$$

$$\lambda_{op}^{arch} = P_{op}^{1fault,arch} E_{op}^{arch} = P_{op}^{1fault,arch} (1 - D_{op}^{arch}) \quad (5.9)$$

$$P_{op}^{1fault,arch} = \prod_i^{arch} P_{op,i}^{1fault} = \prod_i^{arch} (A_{op,i} / A_{1fault/hour}) \quad (5.10)$$

$$A_{1fault/hour} = \frac{1}{FIT} \quad (5.11)$$

$A_{op,i}$ and $P_{op,i}^{1fault}$ for four operators are estimated in Table 5.10 based on 90nm Faraday technology library [59]. Note that the area for routing wires is ignored in the estimation.

	Add	Sub	Sll	Srl
Gate-counts (NAND equivalence)	256	224	128	104
$A_{op,i} (\mu m^2)$	1280	1120	640	560
$P_{op,i}^{1fault}$ per hour	0.128	0.112	0.064	0.056

Table 5.10: Failure rate estimation for four operators

Combined with D_{op} from Figure 5.24, the estimated $MTTF_{op}^{arch}$ under logarithmic scale for four operators on several architecture variants is shown in Figure 5.26. Generally, CGRA architecture naturally provides more reliability than VLIW, which is in turn more robust than RISC architecture.

For the same operator, $MTTF$ increases with both the increasing number of modules in the system and the increasing diversity of the modules. Compared with Figure 5.24, RISC architecture with two diverse modules achieves far less $MTTF$ than VLIW with four identical modules. This is due to the fact that P_{op}^{1fault} for VLIW is significantly smaller than RISC, since more modules in the multiplex system correspond to a lower probability that 1 fault happens in *each* module. The trend is the same in case of CGRA. Among all four operators, Sll shows the highest $MTTF$ which results from its small fault probability due to its size and relatively high design diversity.

Figure 5.26: Mean-time-to-failure of architecture variants

5.3.4 Summary

Design diversity was proposed as a quantifiable metric in reliability evaluation for circuit-level designs. In this work, design diversity is extended as an architecture-level design metric to quantify reliability of different processing architectures in modern SoC. The approach is demonstrated on several architecture variants, where design diversity and system Mean-Time-to-Failure are estimated quickly.

Chapter 6

Architectural Reliability Exploration

In this Chapter, three architecture-level fault tolerant techniques are presented. In Section 6.1 opportunistic redundancy is proposed to protect the algorithmic units of embedded processor with low performance penalty. In Section 6.2 asymmetric redundancy is proposed to protect the memory elements with the feature of unequal error protection based on information criticality. In Section 6.3 error confinement technique is proposed to correct errors in memory with statistical data, which reaches similar protection level with faster performance and less power consumption than traditional techniques.

6.1 Opportunistic Redundancy

Architectural fault tolerant techniques targeting transient faults in mainstream processors can be categorized into spatial and temporal redundancies. Spatial redundancy requires additional hardware for protection, in the form of either information encoding such as ECC [112] for protecting memory elements or simultaneous multiple execution for combinatorial logic such as Triple Modular Redundancy (TMR) [196]. Temporal redundancy involves instruction re-execution on the same hardware such as Simultaneous and Redundantly Threaded (SRT) [154] processors at different clock cycle. Error detection is performed by comparing results from both executions. The processor is either backward corrected by rolling back to its previously correct state or forward corrected with the value from majority votes and ECC decoder.

In contrast to mainstream processors, reliability techniques for embedded processors have reduced design space. Area and power constraints limit spatial redundancy with less amount of execution units and storage buffers while the opportunities for temporal re-execution of instructions are also restricted due to real-time constraint. Expensive hardware units for out-of-order execution and multi-threading are seldom applied. However, redundancy, which still exists, has been neither completely eliminated nor fully utilized due to dynamic hardware usage and compiler complexity. Consequently, underutilized computational resources in embedded processors are expected to be opportunistically used i.e., protect when possible, to achieve best-effort reliability enhancement. Besides, it is cost effective to reduce the granularity of replicated execution into micro-architecture level instead of the whole instruction to achieve less performance penalty.

Contribution In this work, a protection approach is proposed which considers different policies for opportunistic redundant execution. In most aggressive policy, the

protected micro-architecture units are always double executed to ensure computational correctness with large degraded performance. On the other hand, passive policy re-executes only if there exists underutilized resources, which incurs no performance penalty when no errors are detected. Depending on the target application, policies can be chosen in order to balance reliability tolerance with performance. This work also contributes several novel implementation features for reliability enhancement of embedded processors with less area and power overheads. Embedded RISC processor without out-of-order execution capability explores temporal redundancy in ALU units whenever possible. VLIW processor takes advantage of its empty instruction slots for spatial redundant execution. Using ADL based processor design flow, the proposed architectures and protection policies are efficiently explored through high level fault injection. Synthesis results are provided to show the additional area and power cost.

6.1.1 Opportunistic Protection

This section introduces the concept of opportunistic protection. First, an embedded RISC processor is modeled in the form of Directed Acyclic Graph (DAG). Second, the protection level used in this work is derived based on the DAG. The applied policies which facilitate opportunistic protection are explained next.

6.1.1.1 Processor Modeling

The processor modelling methodology from ADL LISA [2] is applied in this work. The complete instruction set is represented by DAGs of LISA operations which describe the instruction behaviors. The directed links between operations show the activation conditions. When operations are associated within pipeline stages, the activation of operations implies the timing information of instructions. LISA operations are implemented as HDL processes during RTL generation. Figure 6.1 shows the DAG for a 5-pipeline stage RISC processor.

The *Decode* operation implements the coding root which activates corresponding sub-operations based on the instruction word from Fetch stage. The sub-operations in Decode stage prepare the operands for calculation in Execute stage. Results from Execute stage are further passed onto memory operations such as *load* and *store* or directly to *Writeback* operation. Note that *alu_ex* is shared by *alu* and *bit* as activated operation which is shown by the solid line. Besides such intra-instruction dependency, the dashed lines show inter-instruction dependencies since other types of instructions depend on the operand values from previous *alu* or *bit* type of instructions.

6.1.1.2 Protection Level

Protection level is critical to the whole design as it directly influences the error coverage and performance overhead. The RMT processors have instruction level protection. To reduce the large performance overhead caused by instruction level duplication, the proposed protection is restricted on micro-architecture level. Figure 6.1 shows that the

Figure 6.1: Directed acyclic graph of embedded RISC processor

error in computational result of *alu_ex* operation directly influences *alu* and *bit* instructions while indirectly affecting other instructions. The importance of protection for *alu_ex* can also be indicated from application profiling. Figure 6.2 shows the instruction distribution averaged from 12 applications of MiBench [71] on the RISC processor model. It is noticeable that *alu* and *bit* account for 51% of all the instructions while *control*, *compare* and *memory* account for 37%. The rest instructions are NOPs which are not relevant for protection.

Consequently, the architecture unit *alu_ex* is double executed with the same inputs to ensure its correctness. However, instead of connected execution, the verified execution is only performed when the *alu_ex* operation is un-activated, which implies that currently other type of instructions are activated in Execute pipeline stage. Additional protection buffers are used to store the operands to be verified, while commit buffers are applied to ensure the correct commit sequence of instructions. When mismatch is detected, the pipeline is rolled back to execute the faulty instruction for error correction. The proposed protection method tries to duplicate the execution of vulnerable architecture units when there is opportunity for doing it. In contrast, instruction level duplication doubles the execution of the complete instruction. For embedded processors with single ALU unit this scheme would manifest large performance overhead.

Figure 6.2: Average instruction distribution for MiBench

6.1.1.3 Protection Features and Policies

Different applications have various requirements in reliability. In Figure 6.3 several features and policies which explore the reliability-performance trade-off are introduced. Figure 6.3a) shows the basic approach for re-execution of *add* instruction in line 2 when line 3 has no ALU instructions. An improved version of this would be the re-execution of several instructions based on a protection buffer which records unprotected instructions. This is shown in Figure 6.3b). The program flow may prohibit connected protections when there are not enough redundancies for protecting previous instructions. In such cases, the following ALU instructions are continuously pushed into the protection buffer and wait for chances of re-execution. However, when the buffer is full or in case of no protection buffer, decision for protection needs to be made in either passive policy such as Figure 6.3c) or aggressive policy in Figure 6.3d). Passive policy ignores the protection of oldest instruction while aggressive policy stalls the processor pipeline for acquiring one clock cycle of in-activated *alu_ex* operation for duplicated execution. With regard to the VLIW processor which has parallel execution syllables, spatial redundancies can be explored as shown in Figure 6.3e) to take advantage of the underutilized execution resources.

6.1.2 Implementation

In this section, the proposed design methodology is demonstrated on embedded RISC and VLIW processors using ADL LISA. Intrinsic temporal and spatial redundancy within processor models are explored to leverage reliability.

Figure 6.3: Protection features and policies

6.1.2.1 RISC Model

PD_RISC_32p6 processor model from Synopsys IPs is improved targeting reliability, which is a fully bypassed 32-bits embedded processor with 6 pipeline stages. With full support of C compiler, applications are compiled to run on the cycle-accurate instruction set simulator generated from LISA compiler. Figure 6.4 explains the general protection flow.

Within each clock cycle, the protection unit begins to check the activation condition of the ALU unit to identify whether redundant operations can be performed. A free ALU unit signals to execute previously oldest recorded instruction from the protection buffer. Results from both executions are compared to decide whether error recovery operation needs to be activated. In case of no error detected, the checked instruction is committed from the commit buffer together with its dependent instructions. In contrast, when the ALU unit is occupied in the current control cycle, decision

Figure 6.4: Flow of protection unit

is made on whether to push current instructions into the protection buffer based on their types. Table 6.1 shows three types of instructions where HARD and SOFT types are recorded for protection. Other instructions are committed without further protection. While the writeback values of SOFT instructions are recorded in the commit buffer directly, HARD type of instructions need further processing based on their protection modes. When the protection buffer is full, aggressive policy tries to stall the pipeline for earning one extra cycle of free ALU usage for re-execution. Otherwise, passive mode overwrites the oldest entry in the protection buffer with the new one without stalling the pipeline.

Figure 6.5 visualizes the protected ALU unit in EX pipeline stage. Several architecture features are explained next.

Protection Buffer The protection buffer stores the opcode, operands and writeback values of the protected instructions. A checkState tag bit is used to indicate whether such instruction has been verified, so that the entry in both protection and commit buffer can be deleted in the next clock cycle. The protection buffer is implemented as

Table 6.1: Handling methods of different instruction types

Types	Definition
HARD	Protected instructions
SOFT	Instructions have data dependency on HARD type
NONE	Other instructions

Figure 6.5: PD_RISC_32p6 processor with protection modification

a circular buffer which wraps around to the beginning when the boundary of buffer entries is reached. The buffer size is adjustable during simulation, which reveals the design trade-off between reliability and area/power. One specific control register is used for management of protection buffer, which periodically gets updated in DC pipeline stage.

Delayed Commit Unit A delayed commit unit is used for keeping the return values of both HARD and SOFT types of instructions. To ensure the correct value is used for subsequent instructions, the data bypass mechanism is adjusted accordingly to establish a new path from commit buffer to the EX pipeline stage. The new path has higher priority in data bypass unit than the one from the register file. Searching data matches is done through register indexes which are stored along with write back values in the commit buffer. The search starts from the newest entity in the commit buffer afterwards to ensure the correct order of value updates.

Error Correction When a mismatch of data values is detected, error is corrected based on the rollback of processor state to the previously correct point. It is achieved by fetching the new instruction based on the value of program counter (PC) stored in the protection buffer. Meanwhile, both the pipeline registers and buffers flush their values.

6.1.2.2 VLIW model

The LT_VLIW_32p5x4 processor model from Synopsys IP is adapted for error protection. The model contains 5 pipeline stages with 4 parallel instruction syllables. The EX stage of each syllable contains individual ALU unit. The major difference from RISC processor is that a VLIW processor contains intrinsic spatial redundancies which can be used for parallel execution. Due to application properties, such redundant instruction slots are not fully explored by the VLIW compiler. The modified architecture is shown in Figure 6.6. Three units are added into the architecture, a centralized *Protection admin*, *Operand routing* and *Post processing* units. The protection buffer and control register are the same as for PD_RISC processor model while a new register *alu_ctrl_reg* is introduced for guiding the mapping of operation and operands among four ALU units in EX pipeline stage. *Operand routing* and *Post processing* units adjust their behaviors accordingly based on the contents in this register.

ALU Control Register The *alu_ctrl_reg* register contains 16 bits which are divided into four fields, each controls a ALU unit respectively. The higher 2 bits of each field represent the operation mode, while the lower 2 bits represent table pointers as shown in Table 6.2. Figure 6.7 shows an example on how this register directs the operation of each ALU. *alu0* and *alu1* contain ALU instructions while *alu2* and *alu3* slots are free in this clock cycle. The value of *alu_ctrl_reg* is set by the *Protection admin* unit in the decode stage. According to Table 6.2, *alu0* duplicates its execution in slot *alu2*, while slot *alu3* re-executes previously recorded operation from protection buffer *entry 1*. The computation from *alu1* is performed and backuped in the protection buffer *entry 2* for later verification. Through such operands mapping, efficient management of redundant resources is achieved to utilize both spatial and temporal redundancy.

Commit Unit For VLIW model the delayed commit unit is not necessary since the instruction will be either executed twice or overwritten by another instruction due to

Figure 6.6: VLIW architecture with protection modification

Table 6.2: ALU control register

Modes	2 MSBs	2 LSBs
Normal execution	00	–
Save to table	01	table write pointer
Fetch from table	11	table read pointer
Fetch from syllable	10	syllable pointer

the protection policies before it enters the write back stage. This is due to the internal parallel architecture of VLIW processors while detailed timing analysis is ignored here in case of simplicity.

6.1.3 Experimental Results

Experiments on two processor models are documented in this section. First, performance overheads are presented based on several embedded applications. High-level fault injection experiments are used to show the protection efficiency. Lastly the synthesis results of proposed design are provided.

Figure 6.7: VLIW control register

6.1.3.1 Benchmarking

Several applications from MiBench are ported to our simulation environment based on the Synopsys Processor Designer tool flow [182]. Figure 6.8 shows the coverages of protected instructions for both policies on PD_RISC processor, where both the protection and commit buffers are configured to store 3 entries of protected operations. It is shown that aggressive policy always achieves more coverage than passive policy. For aggressive policy, performance degradation is incurred by pipeline stall when no free entry exists in protection buffer. The cost of such effect varies due to different properties of applications. For computationally intensive applications such as *sha* and *rijndael* the performance degradation is higher than others. With regard to passive policy, redundant execution is done in an opportunistic way, therefore *no performance overhead* is incurred with decreased instruction coverage. For the VLIW processor the instruction coverage shows similar values which are not presented.

6.1.3.2 Fault injection experiment

To demonstrate the efficiency of error detection and correction of proposed architectures, a state-of-the-art high-level fault injection flow has been adopted [209] which is integrated into LISA based processor design environment. Bit-flip faults are injected into the ALU units randomly during simulation time. *Error Manifestation Rate* (EMR) [42] is used as the method for error evaluation. EMR represents the percentage of detected fault on the program/data memory interface of the processor core. The EMR value indicates the error resilience ability of an architecture.

10,000 experiments are performed to get the EMR value for a fixed count of injected faults. *Sieve* and *IDCT* applications are selected for fault simulation. Figure 6.9 shows the EMR trends with increased count of injected faults for RISC and VLIW processor respectively.

For both processor models, protected architectures in passive mode achieve lower EMR values than unprotected ones. Aggressive mode results in 0 EMR values regard-

Figure 6.8: Instruction coverages and performance degradation

Figure 6.9: EMR with increased count of faults for RISC/VLIW processor

less of the faults count which means all of the injected faults are resilient. Combined with the performance degradation for aggressive policy in Figure 6.8, it reflects the trade-off between reliability and performance. Among different architectures VLIW processor achieves much less EMR than RISC processor under same amount of injected faults. This happens due to the reason that same amount of faults are distributed onto 4 individual ALU units in VLIW processor which reduces the probability of interface error.

Optimization levels of the C compiler have impact on the run-time execution of the generated assembly codes, which affects the efficiency of the passive protection mode. Figure 6.10 shows the optimization impacts on EMR for *Sieve* and *IDCT* applications. The compiler is generated by the CoSy compiler development system [186]. Basic features of optimization levels are briefly explained. For detailed information regarding optimization flags please refer to [142]. It is obvious that effects of optimization levels vary among different applications. Compiler level theoretical analysis to cope with opportunistic protection is beyond the scope of the work.

- O0: No optimization, as applied in Figure 6.8 and 6.9
- O1: Alias analysis and control flow simplification
- O2: Function inlining and object propagation
- O3: Loop level optimizations
- O4: Software pipelining

Figure 6.10: Effects of C compiler optimization levels on EMR for passive mode

6.1.3.3 Synthesis result

Based on the HDL generation flow of Synopsys Processor Designer [182] the proposed architectures are synthesized under 90nm Faraday technology library by using Synopsys Design Compiler, topographical mode. Table 6.3 shows the gate-level synthesis results on area and power overheads for both processor models, where the power values are averaged from the MiBench applications. The major extra area and power contributors are the protection and commit buffers which are implemented as architectural registers. For error detection only, no commit buffer is required for RISC processor so that power and area are smaller than the results in error recovery mode. Error recovery for VLIW processor does not require extra area overhead since commit buffer is not required. VLIW requires relatively more overheads than RISC due to the operand routing unit which connects four ALU units and large amount of register ports to the protection buffer.

Table 6.3 also shows the performance and energy overheads under fault free simulations. The passive mode achieves 0 performance overheads for both architectures whereas the aggressive mode achieves varies overheads among different applications.

Table 6.3: Design overheads for proposed architectures

	Area Overhead		Power Overhead	
	detection	recovery	detection	recovery
RISC	8%+	12%+	17.43%+	25.48%+
VLIW	16.86%+	16.86%+	29.23%+	31.23%+
	Performance Overhead (fault-free simulation)		Energy Overhead (recovery enabled)	
	passive	aggressive	passive	aggressive
RISC	0%	2%-20%+	25.48%+	28.0%-50.6%+
VLIW	0%	2%-12%+	31.23%+	33.9%-47.0%+

6.1.4 Summary

In this work a best-effort design method to increase reliability of embedded processors against soft errors is presented. Redundancies of processor ALU units are explored opportunistically based on several protection features and policies. Taking advantage of the proposed design, protection overhead can be reduced significantly. Synthesis results are provided to show the trade-off between reliability and performance.

6.2 Processor Design with Asymmetric Reliability

Recent research on reliable system design reflects two trends. First, reliability is treated as a *cross-layer design issue* [45]. This stresses the fact that separate error mitigation techniques may result in an over-designed system. The design can and should take the support of architectural and algorithmic error resilience [143]. This approach can reap significant benefits and requires establishment of design metrics and strong understanding of all the design layers. The second trend is to offer *asymmetric reliability*, where different parts of the system are protected unequally [72]. Consequently, the upper design layers, e.g., the application can select the level of reliability statically or dynamically during execution based on the task criticality.

The existing works on reliable storage implementation for embedded processors is essentially a *channel coding* problem in noisy communication. Based on that, error detection and correction methods in unreliable processor microarchitecture can be applied to the storage. The storage include registers, data memory, instruction memory and pipeline registers. The aspect of asymmetric reliability can be further probed for such information theoretic model. Though it is well accepted in the research community that asymmetric reliability leads to more design trade-offs and possibly better Pareto-optimal curves [72, 97, 108], it is yet to be fully explored in the context of embedded processors. While unequal error protections for register file and data memory are proposed by different schemes [108], the reliability issues of instruction words and program memory remain relatively less studied. Instruction-level vulnerability analysis is presented in [161] though, the asymmetry is addressed by software mapping flow only. Naturally, protection of instruction and data demands different techniques. The scope and impact of all aspects of asymmetric reliability are not systematically explored and studied to the best of our knowledge.

Though the availability of rich literature in channel coding aids our work, there are two important challenges, one needs to solve. First, designing hardware-based unequal error detection/correction requires efficient implementation of channel encoder/decoder. A few works [93, 156] discuss applying Hamming codes for on-chip storage but, efficiency of the encoder/decoder in terms of area and runtime is still an important problem. This is particularly needed since, the error detection and correction impacts the critical path of the processor. Second, Borade *et al.* [157] took an information theoretic view on unequal error protection and there exists many papers [103, 155] on applying unequal error-protection in diverse application areas. Interestingly, all of them consider different error probability for different bit-positions and this distribution is the same for all the messages. Here, on the other hand, it is advocated that some instructions in a processor are more important than the others and hence demand more protection - thereby requiring a different view of asymmetry. These problems are addressed in this work.

Contribution In short, the contributions of this work are following.

- An asymmetric processor reliability model.

- Novel schemes for performing reliability trade-off with other design constraints.
- Detailed experiments using a high-level embedded processor design framework.

6.2.1 Asymmetric Reliability

Any practical communication channel is exposed to several noise sources and therefore the message received by the receiver may be different from the message sent by the sender, causing transmission errors. The seminal paper of Shannon [28] laid the foundation of the theory of error-correcting codes (ECC), which deals with how to detect and correct transmission errors at the receiver end. The basic idea is that each message m is transformed into a particular codeword c before being sent. The set of codewords constitute a *code*. If an error is introduced in the communication channel, the received word r would be different from the codeword c . A convenient coding method for binary channels is $[n, k]$ linear block code where the message is divided into blocks of k consecutive bits and each such k -bit message is encoded into an n -bit ($n > k$) codeword.

In Figure 6.11, a classification of asymmetric reliability schemes is proposed. The unequal error protection for different channels in a microarchitecture can be either static or dynamic. For statically asymmetric reliability, the error protection schemes are fixed before program execution. For dynamic, it can be altered depending on the varying criticality of tasks/data/instructions. For both static and dynamic asymmetric reliability, the error protection can be distributed across or within different messages. If only the higher order bits of a data are protected for every data then, the reliability distribution is bit-wise. If the protected regions vary for different data/instruction then, it is termed as message-wise asymmetric reliability. For example, a static error detection and correction framework is investigated for the processor instruction set using message-wise asymmetry. The program memory is viewed as a channel, where linear codes are needed.

It turns out that the general decoding problem of linear codes is NP-complete [52]. Though there are particular classes of linear-time encodable and decodable error-correcting codes [180], the time required to decode a codeword may adversely affect the critical path of the microarchitecture. Hamming code [146] is therefore utilized which can be decoded in a single cycle.

For each integer r , there is a Hamming code with codeword length $n = 2^r - 1$ and message length $k = 2^r - r - 1$. It has a $k \times n$ generator matrix G so that for a message (considered as a row-vector) u , the corresponding codeword is given by $v = uG$. Note that $r = n - k$ denotes the number of parity bits when G is written in systematic form (i.e., has a submatrix I_k in the left part). The $r \times n$ parity check matrix H contains all the $2^r - 1$ non-zero r -bit binary vectors and the syndrome vector $s = v^T H$ gives the position of the error in case of single-bit error. If $s = \mathbf{0}$, it indicates that no error has occurred. An extra parity bit can be added to form what is called an *extended Hamming code* that provides an additional facility of detection of double bit-error. However, for proof of concept, only single bit-error correcting Hamming codes are used. In fact,

Figure 6.11: Asymmetric encoding and decoding

our asymmetric reliability framework is generic in the sense that any error-correction component (that is fast enough) can be used as its subsystem. For memory word size of 32 bits, $r = 6$ parity bits are needed to correct a single bit error, leading to 38-bit codewords. This scheme is denoted by $H_1[38, 32]$.

6.2.1.1 Divide and Conquer Method for Higher Bit-Error Correction

For double-bit error-correction, one could use BCH codes as in [140] or LDPC/turbo code as in [144]. However, the state-of-the-art decoding algorithms and circuits for these codes are suitable for memory protection but are not fast enough to be used in the instruction pipeline.

Note that the total number of possible errors in a 32-bit message is $2^{32} - 1$. Out of this, the direct application of Hamming code, i.e., the scheme $H_1[38, 32]$ covers $\binom{32}{1} = 32$ cases. To tackle higher bit errors, a novel method of dividing the message into multiple segments and apply Hamming encoding and decoding on each part parallel is proposed. For example, when a 32-bit word is divided into two halves, each of 16 bits and apply Hamming code with $r = 5$ parity bits on each half, then

$((\binom{16}{0} + \binom{16}{1}))^2 - 1 = 288$ cases can be covered. This scheme is denoted by $H_2[42, 32]$. Similarly, the 32-bit word can be divided into four parts each of 8-bits and apply a Hamming code with $r = 4$ parity bits on each part to achieve partial four-bit error-correctability. This scheme is referred as $H_4[48, 32]$ and covers $((\binom{8}{0} + \binom{8}{1}))^4 - 1 = 1295$ cases. Though the schemes $H_2[42, 32]$ and $H_4[48, 32]$ cannot correct arbitrary double or four bit-errors, by further division into segments of size 4 bits, 2 bits and so on, in the limit, one can correct any number of errors in the whole word. This scheme is called as *Divide and Conquer Hamming* or DCH.

Suppose a $m = kl$ -bit message is divided into l parts each of k bits and apply a single-error correcting Hamming Code on each part. For a k -bit message, the number of parity bits needed is given by the minimum integer r_k such that, $2^{r_k} - r_k - 1 \geq k$. The total number of parity bits for all the l parts are given by $r_k l$. The resulting partial l -bit correcting DCH scheme is denoted by $H_l[kl + r_k l, kl]$. Table 6.4 shows the typical parameter values for different choices of l and k assuming $m = kl = 32$.

Table 6.4: DCH schemes from different message partitioning

l	k	r_k	$r_k l$	DCH Scheme
1	32	6	6	$H_1[38, 32]$
2	16	5	10	$H_2[42, 32]$
4	8	4	16	$H_4[48, 32]$
8	4	3	24	$H_8[56, 32]$
16	2	3	48	$H_{16}[80, 32]$
32	1	2	64	$H_{32}[96, 32]$

The following result states how many out of total $2^m - 1 = 2^{kl} - 1$ non-zero error vectors are corrected by such a scheme.

Theorem 6.2.1 $H_l[kl + r_k l, kl]$ scheme can correct $(k + 1)^l - 1$ many different errors in kl -bit messages.

Proof: In each of the k -bit part, there can be $\binom{k}{1} = k$ single bit errors. Adding to this the case of no error, each of the l parts can contribute $k + 1$ cases, giving a total of $(k + 1)^l$ cases. But this include the case where no error occurs in each part. Thus, excluding the all zero error-vector, the total number of errors covered is given by $(k + 1)^l - 1$.

It is easy to see that with $m = kl$ fixed, as the number l of parts increases, the individual part-length k decreases and the quantity $(k + 1)^l - 1$ increases. In the limit, when there are m parts each of length 1, the resulting $H_m[3m, m]$ scheme can correct all the $2^m - 1$ error vectors. This is exactly the point of the following corollary.

Corollary 6.2.2 $H_m[3m, m]$ scheme can correct all the $2^m - 1$ errors in m -bit messages.

Proof: In Theorem 6.2.1, when $l = m$ and $k = 1$, there exists $(k + 1)^l - 1 = 2^m - 1$.

There are two main points in which the proposed scheme for instruction word differs from the traditional coding schemes in communication systems.

- First, in coding theory, increment in block size tends to increase the coding efficiency. However, for instruction word, increment in block size would increase the error-correction time as well. Our method works on multiple blocks in parallel and therefore decoding the whole word takes the same time as decoding a single block.
- Secondly, the proposed DCH coding cannot be applied to general communication systems. The reason is in typical communication scenario, the issues of framing, packetization, synchronization and real time constraints force the receiver to decode the incoming bitstream as and when it arrives. The receiver cannot afford to wait for the entire message to arrive and then perform divide and conquer. On the other hand, in instruction memory, the whole word is available for applying the divide and conquer technique.

6.2.2 Asymmetric Reliability Exploration

With the proposed techniques multiple sets of experiments are performed using the environments described in this section. First, the efficiency of the proposed Hamming encoder/decoder is studied together with its impact for different levels of protection. Second, a comparative study between static and dynamic protection modes is done. Third, the effect of different asymmetric reliability techniques for VLIW processor is investigated. Finally, a comparison of message-wise and bit-wise asymmetric reliability is performed.

6.2.2.1 Experiment Setup

For all the experiments, Synopsys Processor Designer [182], version 2012.06-SP1 is used. The processor descriptions, used in this work, are developed from the basic processor templates provided with the tool. The first processor, denoted *LT_RISC_32p5*, is based on a RISC instruction set with 5-stage pipeline. The second processor, denoted *LT_VLIW_32p5x4*, contains a 5-stage pipeline and can have up to 128-bit instruction word distributed over four parallel 32-bit instruction slots. The instruction-set supports the basic RISC instructions as available in *LT_RISC_32p5* model. Both the models are fully synthesizable and provides readily available retargeted C Compiler. Changes to the microarchitecture can be easily performed for both the models, with fast automatic retargeting of the software tools and generation of synthesizable RTL. For gate-level synthesis, Synopsys Design Compiler, version G-2012.06, targeting a 90nm technology CMOS technology library is used. The applications for our experiments are chosen from different types of embedded applications including multimedia and wireless algorithms.

Technique in [209] is applied to efficiently inject fault in cycle-accurate instruction set simulator generated from Processor Designer. A user friendly GUI is designed for faults configuration. The *Error Manifestation Rate* (EMR) [42] is used as a metric for the evaluation of fault simulation. For a set of fault injection experiments, *EMR* is defined as the percentage of experiments which detects error on the memory interfaces to which the faults propagate. Normally, *EMR* value increases with the duration and number of injected faults. For the same fault duration and number, larger *EMR* value indicates less reliable component.

Figure 6.12: ECC encoding and decoding

6.2.2.2 Efficient ECC for Message-wise Protection

Figure 6.12 shows the work flow for asymmetric protection for program memory. The application is initially compiled, assembled and linked to prepare the binary executable. The instructions are encoded into code words for different ECC modes. The code words are loaded into multiple ECC memories. A hardware-based decoder which detects and corrects possible faults in the program memory is integrated into the fetch pipeline stage of the processor. To facilitate runtime asymmetric protection, the decoder selects the code words from several ECC memories to enable different level of protection. The protected instruction word is then forwarded onto the next pipeline stage.

The current modes of protection for 32 bits instruction words include H_1 [38,32], H_2 [42,32] and H_4 [48,3] that can achieve a maximum of 1, 2 and 4-bit error-correction respectively. To demonstrate the protection efficiency by applying different modes, the fault injection experiments are carried out with bit-flip faults injected into both program memory and ECC memories randomly. The example application running on the processor is *sieve of Eratosthenes* which requires a program memory size of 512 Bytes. Figure 6.13 shows the EMR for different protection modes with increasing number of bit-flip faults. Each evaluation point is averaged from 1000 experiments while each experiment lasts for 1,200 clock cycles. With increased protection level

Figure 6.13: EMR with different protection modes (Sieve application on RISC processor)

the EMR values drop at the same evaluation point, which shows an increased protection ability. All EMR values increase with increased number of faults where higher protection level shows better EMR performances.

Table 6.5 shows the trade-off between power and area for the RISC processor with different protection levels. All the architectures are synthesizable with a maximal frequency of 200MHz. The power consumption increases slightly with increased protection level. The RISC core area reflects an inverse trend since the decoder which decodes the complete 32-bit instruction costs more than twice the area of 16-bit decoder. The same trend between the 16-bit and the 8-bit decoder is also observed. The ECC memory size increases with increased protection level.

Table 6.5: Performances for different protection modes

Protection Level	Area (KGates)	ECC Size (Bytes)	Power (mW)
No Protection	27.53	0	9.03
H_1 [38,32]	27.94	96	9.26
H_2 [42,32]	27.93	160	9.28
H_4 [48,32]	27.92	256	9.30

6.2.2.3 Static and Dynamic Asymmetric Reliability

Experiments have been carried out to investigate the impact of varying instruction reliability. Static and dynamic approaches are proposed in the assignment of instruction criticality. The static approach based on the observation that different instructions reflects different criticality. For instance, *branch* is more critical than *alu* instructions due to the effect of altering the program flow. To characterize the criticality of individual instructions, simulations are performed by injecting random faults onto the same set of instructions with random operand values. Figure 6.14 collects the simulation results where Instruction Vulnerability Factor (IVF) are characterized. We estimate IVF by the instruction error rates (IER) defined in Section 5.1.2. For a specific instruction, the IER values for all logic operations on the activation path of such instruction is averaged to calculate the IVF, where each IER value is collected from 10,000 single bit-flip fault injection experiments. Note that, different techniques of obtaining the instruction criticality can be conceived, including a user-driven criticality assignment.

Figure 6.14: Static instruction criticality assignment

The dynamic approach assigns the protection level based on the situation of run-time fault detection, which reduces the power consumption of the ECC decoder when

less faults are detected. Figure 6.15 shows the approach in the form of a Finite State Machine (FSM). Initially protection level one is assigned to all the instructions. If a fault is detected the processor updates to level three for the following instructions, which lasts for at least 10 clock cycles. After that time period, decision is made again based on whether faults are detected and protection may reduce to level two. If a fault is detected at level two, the protection level is increased for the following instructions.

Experiments are performed to compare the protection efficiency of both approaches. The ECC decoder which supports three modes are integrated into the RISC processor. For static approach, the coding of instructions are indexed with two ECC mode bits according to Figure 6.14 for runtime mode selection. For dynamic approach a state machine module according to Figure 6.15 is designed. A flag register is used to switch between static and dynamic modes.

Figure 6.16 shows that static approach excels the dynamic one when several single bit-flip faults are injected randomly into whole memory blocks. However, when adjacent instructions in memory array are bit-flipped, dynamic approach achieves better protection since high protection level covers all adjacent instructions. The dynamic approach contributes especially when the faults happen inside loop kernels where sequential instructions are defended using maximal protection.

Figure 6.15: FSM for dynamic, asymmetric reliability

6.2.2.4 Asymmetric Reliability for VLIW Processor

VLIW architectures provide further opportunities for asymmetric protection. Instructions generated by the VLIW compiler leave certain slots empty, when sufficient instruction-level parallelism is not detected in the application. These slots are potentially to be utilized to place the parity bits for protecting the instruction words. To

Figure 6.16: Comparing static and dynamic protection

simplify the design complexity, two slots are fixed for protection when no instructions reside in. Figure 6.17 visualizes the scheme.

To enable asymmetric protection for VLIW instructions, each slot is filled with parity bits of different modes. Ideally in a 32-bit instruction slot, all the three types of parity bits corresponding to $H_1[38, 32]$, $H_2[42, 32]$ and $H_4[48, 32]$ can be exactly filled in. However, in our target VLIW model one bit for each slot is reserved to mark the end of VLIW instruction. Therefore, parity bits from two ECC types are used. In each slot one extra bit is used to indicate whether it contains an instruction or parity bits.

As it can be observed from Table 6.6, both the area and the power consumption increases with increased protection levels. In this experiment, the parity bits for each 32-bit instruction is generated and filled in the VLIW slots. This required us to introduce empty slots in the application, when there is none. This resulted in an increase of the application runtime. To better study the effect of application runtime increase, two different protection modes are conceived. In the first mode, named *opportunistic* [199], parity bits for instructions are fitted in, only if empty slots are present. For another, *compulsory* protection mode, parity bits are introduced for all the instructions.

Table 6.6: Reliability vs. power/area trade-off

Protection Level	Area (KGates)	Power (mW)			
		Cordic	Sobel	FFT	CRC32
No Protection	79.09	4.93	4.89	4.91	5.00
$H_1[38, 32]$	79.85	5.18	5.13	5.17	5.25
$H_1[38, 32]$ and $H_4[48, 32]$	80.49	5.40	5.34	5.38	5.45

The EMR results of this experiment are shown in Figure 6.18 where the corresponding application runtime is provided in Table 6.7. Note that larger differences in

Figure 6.17: ECC in VLIW slots

clock cycles between two protection modes lead to larger gap of *EMR*. This shows an interesting trade-off between available parallelism, area, runtime, power and reliability.

Table 6.7: Application runtime for various VLIW protection modes

Application	Cycles		Increase (%)
	Opportunistic	Compulsory	
Sobel	2671	2748	2.8
Cordic	1074	1317	22.63
CRC32	32278	32279	3e-5
FFT	1433	1440	0.49

6.2.2.5 Bit-wise Asymmetric Reliability for Data Memory

In contrast to the instruction words, data words can be protected with bit-wise protection schemes, e.g., higher order bits can play a more important role in correct computation than lower order bits. Based on such principle bit-wise asymmetric encoding is explored for data memory. Figure 6.19 shows the proposed flow where, an asymmetric encoder is integrated into the *store* pipeline stage. The parity bits are stored into an ECC data memory in parallel with store instruction. Decoding and correction for the same data is performed in the stage *load*.

To compare the protection efficiency, bit-wise symmetric encoding scheme is also implemented and tested where 4 bytes within 32 bits use the same level of encoding.

Figure 6.18: EMR for different protection modes: VLIW

To compare the performance of asymmetric against symmetric reliability distribution for data memory, two different applications are subjected to fault injection in the data words and traced against golden simulation. Figure 6.20 shows the EMR trends, which reflect that asymmetric protection wins over symmetric protection under the same evaluation points. It is interesting to note that in general, the asymmetric mode of reliability provides higher protection against bit errors though, at a slightly increased area cost (18 instead of 16 parity bits). However, the EMR trends for asymmetric and symmetric mode are very close for Cordic application, indicating that exact choice of reliability mode is application-specific.

6.2.3 Summary

In this work, an asymmetric design framework for exploring reliability of embedded processors is adopted. Based on that, different design variants for multiple architectures are explored to trade off reliability for other performance metrics.

Figure 6.19: Bit-wise asymmetric encoding

Figure 6.20: Comparing symmetric and asymmetric bit-wise protection

6.3 Approximate Computing with Statistical Error Confinement

The aggressive shrinking of transistors have made circuits and especially memory cells more prone to parametric variations and soft errors that are expected to double for every technology generation [25], thus threatening their correct functionality. The increasing demand for larger on-chip memory capacity, predicted to exceed 70% of the die area in multiprocessors by 2017 is expected to further worsen the failure rates [168], thus indicating the need for immediate adoption of effective fault tolerant techniques.

Techniques such as Error Correcting Codes (ECC) [55] and Checkpointing [50] may have helped in correcting memory failures, however they incur large area, performance and power overheads ending up wasting resources and contracting with the high memory density requirements. With an effort to limit such overheads, recent approaches exploit the tolerance to faults/approximations of many applications [33] and relax the requirement of 100% correctness. The main idea of such methods is the restricted use of robust but power hungry bit-cells and methods such as ECC to protect only the bits that play a more significant role in shaping the output quality [208] [107]. Few very recent approaches exist also that extend generic instruction sets with approximation features and specialized hardware units [58] [192] [164]. Although such techniques are very interesting and showcase the available possibilities in certain applications, they are still based on redundancy and have neglected to exploit some more fundamental characteristics of the application data.

Contribution In this work, the state-of-the-art is enhanced by proposing an alternative system level method for mitigating memory failures and presenting the necessary software and hardware features for realizing it within a RISC processor. The proposed approach, instead of adding circuit level redundancy to correct memory errors tries to limit the impact of those errors in the output quality by replacing any erroneous data with the best available estimate of those data. The proposed approach is realized by enhancing a common programming model and a RISC processor with custom instructions and low cost hardware support modules. The low overhead error mitigation ability of the proposed approach is demonstrated by on the different algorithmic stages of JPEG and comparing with the extensively used Single Error Correction Double Error Detection (SECDED) method. Overall, the proposed scheme offers better error confinement since it is based on application specific statistical characteristics, while allowing to mitigate single and multiple bit errors with substantially less overheads.

6.3.1 Proposed Error Confinement Method

Assume that a set of data $d \in \mathcal{D} = \{d_1, \dots, d_K\}$ being produced by an application are distributed according to the probability mass function $P_d(d_k) = \Pr(d = d_k)$. Such data are being stored in a memory, which is affected by parametric variations causing errors (i.e. bit flips) in some of the bit-cells. Sure errors eventually result in erroneous data leading to a new data distribution $\bar{P}_{d_k}$. The impact of such faults can be quantified by using a relevant error cost metric which in many cases is the mean square error (MSE) defined as

$$\mathcal{C}(\bar{d}) \triangleq \mathbb{E}\{(d - \bar{d})^2\} \quad (6.1)$$

with the expectation taken over the memory input d . The proposed method focuses on minimizing the MSE between the original stored data d and the erroneous data $\bar{d}$ in case of a-priori information about the error $\mathcal{F}$ through an error-mitigation function $d^* = g(\mathcal{F})$ which can be obtained by solving the following optimization problem:

$$d^* = g(\mathcal{F}) \triangleq \arg \min_{\bar{d}} \mathcal{C}(\bar{d} | \mathcal{F}). \quad (6.2)$$

where,

$$\mathcal{C}(\bar{d} | \mathcal{F}) \triangleq \mathbb{E}\{(d - \bar{d})^2 | \mathcal{F}\} \quad (6.3)$$

Basic arithmetic manipulations show that the resulting correction function is given by $g_{\text{MMSE}} = \mathbb{E}\{d[n] | \mathcal{F}\}$. This essentially corresponds to the expected value of the original fault-free data. Such expected values can be eventually determined offline through Monte-Carlo simulations or analytically in case that the reference data distribution is known already as in many DSP applications. Note that the above function depends on the applied cost metric that is relevant for the target application and other functions may exist that can be found by following the above procedure. In this work, MSE is focused on which is relevant for many applications and especially for the case study discussed later.

6.3.2 Realizing the Proposed Error Confinement in a RISC Processor

The proposed Error-Confinement function requires a scheme for detecting a memory error in order to provide the needed a-priori information $\mathcal{F}$ and a look up table for storing the expected reference values, which are to be used for replacing the erroneous data. Obviously the realization of such a scheme in a processor requires i) the introduction of custom instructions and ii) micro-architectural enhancements which are discussed next.

The proposed enhancements are implemented on the RISC processor core IP from Synopsys Processor Designer [182], which consists of five pipeline stages as depicted in Figure 6.21, supports mixed 16/32 bits instructions, while the HDL implementation of the core is fully synthesizable. Note that for the detection of an error required in the proposed scheme, a single parity bit is used within each word which is sufficient for detecting a single error. By doing so the required overhead is limited as opposed to ECC methods that require the addition of several parity bits for the detection and correction of a single or more errors.

6.3.2.1 Custom Instructions

At the assembly level 4 new instructions are introduced, which can be used either in standalone assembly or be embedded as inline assembly in a high-level language such as C/C++. To begin with the start address and the word size of the memory block which is going to be protected need to be specified. It indicates the place in the look up table (LUT) as well its size, where the expected value to be used in case of an error is stored. To this end the following instruction is introduced: *set_data @{data_start} @{data_size} @{lut_start} @{lut_size}*

in which all arguments are provided using general purpose registers.

Figure 6.21: Microarchitecture of RISC processor with enhancements for statistical based error confinement

Furthermore, the instruction *chk_load @{dst} @{src} @{index}* is introduced for statistically confining the error in specific memory blocks while performing memory reads. In particular, before reading the protected data, this instruction detects any error within the read data in the register @{src} and in case i) of an error it replaces the erroneous data with the reference expected value stored in the position @{index} of the LUT and loads the value into the register @{dst}, while ii) in case of no error the register @{dst} is assigned directly to the correct value kept in the register @{src}.

Finally, to enable the protection of specific memory write accesses the instruction *en_parity* is introduced as well as the instruction *dis_parity* for disabling the protection of any data if needed. The above instructions are incorporated in the newly constructed LLVM based C compiler (through the use of Synopsys Processor Designer [182]), which supports instruction set extensions using inline assembly.

6.3.2.2 Micro-Architectural Enhancements

The introduced instructions require the enhancement of the microarchitecture of the target RISC processor with customized modules which are highlighted in Figure 6.21. The detailed functionality of the logic functions within each module in each pipeline stage is described in detail in Figure 6.22.

QoS_DC	<p>DC stage module to decode 4 new instructions in ISA:</p> <ol style="list-style-type: none"> 1. en_parity_st: Enable parity encoding 2. dis_parity_st: Disable parity encoding 3. set_data: Register protected data range and reference LUT 4. chk_ld: Check the data using parity value
Gen_parity	<p>EX stage module, implement following operation:</p> <ol style="list-style-type: none"> 1. parity in MEM stage is activated in case the stored data lies in the protected range
QoS_EX	<p>EX stage module, implement following operation:</p> <ol style="list-style-type: none"> 1. chk_ld_ex: Activated by chk_ld, verify parity bit of the incoming data. In case of mismatch, activate chk_ld_mem in MEM, otherwise activate chk_ld_wb_correct
QoS_MEM	<p>MEM stage module, implement following operations:</p> <ol style="list-style-type: none"> 1. parity: Main parity encoding logic. Each 32 bits data generates 1 bit parity. Encode 32 words as 1 word parity, activate paritiy_st logic in WB stage 32 bits parity are ready 2. chk_ld_mem: When a parity mismatch is detected, get the address of corresponding LUT element and issue memory read. Activate chk_ld_wb_wrong in WB stage
QoS_WB	<p>WB stage module, implement following operations:</p> <ol style="list-style-type: none"> 1. parity_st: Store 32 bits parity as one data word into data memory 2. Chk_ld_wb_wrong: In case of parity mismatch, return the reference value from LUT 3. Chk_ld_wb_correct: No parity mismatch, return the input value from instruction
QoS_Registers	<p>Special registers in RegisterFile,</p> <ol style="list-style-type: none"> 1. set and re-set the processor state for load check. 2. Record the location for LUT and runtime offset to access the reference data.

Figure 6.22: Introduced modules and their functionalities

6.3.3 Case Study and Statistical Analysis

6.3.3.1 Case Study - JPEG

To demonstrate the efficacy of the proposed scheme, JPEG is used as case study, which is a widely used lossy compression technique of digital images that became a popular application example among error resilient techniques. JPEG consists of several stages including color space transformation and down sampling. This work focuses on the subsystem shown in Figure 6.23 which consists of four major procedures. In particular an input image of size 512×512 is decomposed into 4,096 matrices of the size 8×8 . Then each matrix is being processed individually by the 2D Discrete Cosine Transformation (2-D DCT) [68] that essentially transforms the image into the frequency domain producing the DCT coefficients as output which are then finally being quantized. For the reconstruction of the image De-quantization and 2D Inverse Discrete Cosine Transformation (2-D IDCT) are applied. In general, the quality of the output image compared to the original one is evaluated using the peak signal to noise ration (PSNR) [85] and a typical PSNR value for a lossy image is 30 dB.

Figure 6.23: Subsystem in JPEG application

$$\begin{bmatrix} -242.538422 & -1.565797 & 0.646120 & 1.169245 & -0.020233 & 0.940231 & 0.027707 & 0.733925 \\ -0.222834 & -0.108308 & -0.780093 & 0.133814 & 0.039231 & 0.062401 & 0.025521 & -0.004177 \\ -0.649843 & -0.512487 & 0.228183 & 0.353566 & 0.000773 & 0.069678 & 0.012114 & -0.001641 \\ -0.641493 & 0.030510 & 0.412436 & 0.099906 & 0.021144 & -0.039063 & -0.002198 & 0.004568 \\ 0.017609 & -0.038111 & -0.065816 & -0.025756 & -0.081390 & 0.005183 & -0.007455 & -0.005406 \\ -0.430538 & -0.021713 & 0.046092 & 0.007092 & -0.002983 & -0.000642 & -0.005067 & 0.005527 \\ 0.084360 & 0.018600 & -0.003267 & -0.000333 & -0.004912 & 0.004006 & 0.001126 & -0.009209 \\ -2.868614 & 0.005601 & -0.002588 & 0.005061 & 0.000778 & -0.002361 & -0.002779 & -0.003040 \end{bmatrix}$$

(a) Average 8x8 matrix after DCT operation

$$\begin{bmatrix} -15.150879 & -0.142334 & 0.065918 & 0.074463 & -0.002197 & 0.003174 & 0.000000 & 0.000000 \\ -0.015137 & -0.008301 & -0.055176 & 0.002930 & 0.000000 & 0.000244 & 0.000000 & 0.000000 \\ -0.045898 & -0.039795 & 0.015381 & 0.003174 & -0.001953 & 0.000000 & 0.000000 & 0.000000 \\ -0.035645 & 0.001953 & 0.018311 & 0.002441 & 0.000244 & 0.000000 & 0.000000 & 0.000000 \\ -0.002197 & -0.002197 & -0.000488 & -0.000244 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ -0.002686 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \end{bmatrix}$$

(b) Average 8x8 matrix after Quantization

Figure 6.24: Reference matrix for DCT and Quantization Coefficients

6.3.3.2 Statistical Analysis of JPEG

Following the steps of the proposed approach, the different stages of JPEG are statistically analysed by performing several simulations with different images. Simulations show that the output matrices of DCT and quantization share a similar pattern; the elements at the top-left corner of both DCT and quantization output matrix are larger in magnitude compared to the rest which in most cases are close to zero. Figure 6.24 shows the expected value of each element in the DCT and quantization output matrix after averaging their values across 4,096 individual matrices for over 10 images. Such values are used as the reference expected values for replacing the erroneous data in case of a detected memory error in the approach. Note that these values are stored in a LUT that was described in Section III.

```

int imageEx[SIZE][SIZE];
int imageData[ROW_SIZE][COL_SIZE];
int imageExtended[ROW_SIZE][COL_SIZE];
// reference LUT containing generalized data
int lut_imageEx[8][8]={780, -1, 0, 1, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0 },
                      {-2, 0, 0, 0, 0, 0, 0, 0 }};

int DiscreteCosine(int imageData[SIZE][SIZE], int imageEx[SIZE][SIZE])
{
    .....
    asm("enable_parity");    // turn on store protection
    imageEx[i1][j1]= (int)sum;
    asm("disable_parity");    // turn off store protection
    .....
}

int Quantization(int imageEx[SIZE][SIZE], int imageExtended[SIZE][SIZE])
{
    .....
    int src = imageEx[i1][j1];
    int dst;
    asm("check_value @dst, @src, (8*i+j)"); // automatic correction
    imageExtended[i1][j1]= (dst/quant[i][j]);
    .....
}

int main()
{
    .....
    // register range of protected data using reference LUT
    asm("set_data @imageEx, 262144, @lut_imageEx, 64");
    DiscreteCosine(imageData, imageEx);
    Quantization(imageEx, imageExtended);
    .....
}

```

Figure 6.25: Programming example with custom instructions for DCT

6.3.4 Results

6.3.4.1 Experimental Setup

The RISC processor is modified and enabled the injection of bit flips in the memory locations storing the images and intermediate results of the JPEG. Note that no errors are injected on instruction cache and other registers which are assumed to be adequately protected.

For detecting errors each of the 32-bit data of the application is encoded with a single parity bit which is sufficient for detecting a single fault. Following the proposed method, the new instructions were used as inline assembly to describe JPEG as shown in Figure 6.25. In this example an array containing the reference expected values for the DCT coefficients is defined. Within the DCT function, before performing a store to the memory, parity encoding is enabled, which is turned off after a write-store operation. Within the quantization function, the load check is performed whenever a value is read out from the array where the DCT coefficients are stored for replacing it with the relevant expected value in case of an error.

The above code was compiled and executed on the modified processor and the performance, power and quality were measured under different error rates as discussed next. Note that for comparison a similar infrastructure is replicated by using a conventional SECDED Hamming code scheme $H[38,32]$ for the protection of the specific memories (protected by the proposed scheme), which requires 6 parity bits for encoding each 32 bit memory word.

6.3.4.2 Evaluation of Quality

Figure 6.26 shows the output images and corresponding PSNR values with different numbers of injected bit-flips according to typical error rates in 65nm process technology. The results show that in case of 800 and 1000 bit-flips, the output image is degraded by 7.6% and 41.2% compared to the error free case.

The reason for such a large degradation in case of 1000 bit-flips is that two bit-flips in the same data word are allowed which cannot be detected by the single bit parity. Careful examination of the simulations indicated that some of such double bit-flips affected words that relate to the first 20 DCT coefficients of the 8×8 matrix (remember there are 4092 such matrices in each image). As other works have also shown such coefficients control almost 85% of the overall image quality and thus if they get affected by errors and these are not tackled by any means as in this case, they lead to significant quality degradation.

As mentioned the quality achieved by the proposed approach is compared with a SECDED ECC. Figure 6.27 shows the obtained results in case of protecting the output of DCT and quantization coefficients with the two schemes under different number of single bit-flips. It is observed that as the number of the injected single bit-flips increases, the output quality (in terms of PSNR) achieved by using the proposed approach is slightly less than that achieved by using the ECC scheme. This can be attributed to the fact that in some cases the correct value of the erroneous data that is being substituted by the expected value may indeed lie in the tale of the distribution and thus may be far from the used reference expected value. In these cases the replacement will not be as accurate and thus the quality achieved by the proposed approach may not be as perfect. In any case the proposed approach tries to confine the impact of memory errors by essentially approximating erroneous data with their expectation and sometimes such an approximation may not be as good. However, note that the proposed approach still achieves to provide output images with PSNR above 36 dB even under 800 bit-flips, closely approximating the error free image.

It is observed that above 800 bit-flips (when double bit-flips are allowed in each word) both methods fail to produce a good enough image, since neither scheme is able to detect and mitigate from multiple bit-flips in single data word. In particular, on one side the SECDED ECC intrinsically cannot correct more than one error in a word and on the other side the single parity bit used to in the proposed scheme cannot detect two bit-flips in a word and thus it does not engage the replacement of the erroneous data.

Figure 6.26: Output images under different schemes of error injection

The results reveal also a different aspect in the JPEG application. In particular, it is observed that in case of more than 800 bit-flips when double bit-flips are taking place in each word then any untreated error in quantization coefficients are far more severe (causing large quality degradation) compared to untreated errors in DCT coefficients. This can be attributed to the sparse nature of the quantization coefficients (i.e. most of them are zero) and the fact that any untreated error will significantly alter the expected distribution of these data.

In addition to the above experiments, the ability of the proposed approach to address multiple bit-flips in a single data word is also evaluated by replacing it with the expected reference value. Figure 6.27c) shows the achieved PSNR under different number of bit-flips in each word. It is observed that the proposed scheme helps to obtain a PSNR of more than 38dB (for the particular image) in case of odd number of faulty bit cells (when the parity bit can detect the error) while the PSNR degrades a lot in case of even number of faulty bit cells (which cannot be detected by a single parity bit). On the contrary, note that the SECDED ECC even with the use of 6 parity bits fail to address any number of multi bit-flips requiring more complex ECC schemes with much more parity bits. All in all, the proposed approach even with the use of single parity bit is able to address adequately the cases of odd multi bit-flips in a single word. The addition of another parity could be employed to improve the capability of

Figure 6.27: PSNR under no protection, proposed scheme and ECC

error detection which is left for future experimentation. The essential conclusion is that the replacement of erroneous data with an expected value suffices to confine the impact of single or even multi memory bitflips.

6.3.4.3 Performance and Power Results

The proposed enhanced processor is synthesized in 65nm Faraday technology and the power, performance and area results compared to the original processor are shown in Table 6.8. Note that the reference processor in this case does not employ any protection scheme and the results in this paragraph try to reveal the overheads involved in enabling preferential protection of specific parts of a memory with special instructions as well as the cost of the proposed data replacement scheme. It can be observed that the performance is decreased by only 4.2% but the instruction extensions for realization of the proposed scheme by a generic programming environment have resulted in large power and area overheads. The extra logic and registers for specifying the protected memory addresses (which is a unique and desirable feature in current error resilient systems enabled by the proposed extensions), the added LUT and the 1-bit parity encoding are responsible for such overheads. However, note that implementing

the same instruction extensions by using six parity bits as needed by a H(38,32) ECC will result in much larger overheads.

Table 6.8: Results for the proposed architecture extensions compared to the reference unprotected processor

	Area (NAND equiv.)		Power (μ Watt)		Critical path (ns)
	Comb.	Seq.	Dynamic	Leakage	
Original	11789	6187	206	65	6.12
Proposed extensions	26519	10663	349	124	6.38
Increase (%)	124.9	72.3	69.4	90.8	4.2

To compare with the SECDED ECC, the total time required for executing the JPEG application on a processor instance that involves the proposed scheme and on another that implements the ECC are presented. Figure 6.28 depicts the overall execution time of the JPEG application after processing images of different sizes from 8×8 till $1,024 \times 1,024$ and correcting randomly injected errors (in same locations) with ECC and the proposed scheme.

For small images both methods take similar time since the modules other than the ones shown in Figure 6.23 dominate the execution time. For images larger than 64×64 , ECC takes significantly longer time compared to proposed scheme. In particular, for an image of size $1,024 \times 1,024$, ECC takes $3.5\times$ more time than the proposed scheme. Note that such overhead will further increase for larger images and more injected errors.

Although the architecture extension achieves large power overhead, the energy consumption ratio between proposed approach and ECC reduces as image size grows, which is illustrated in Figure 6.29. This is because ECC takes longer time to finish. Starting from image size of 128×128 , the proposed approach consumes less energy than ECC, while the energy benefit increases even further for larger images.

Another interesting comparison to discuss is the difference in terms of memory usage. As indicated in Figure 6.28 the proposed approach uses far less memory compared to SECDED ECC scheme which incurs 18.75% memory overhead in each protected data word. In particular, for an image of size $1,024 \times 1,024$, ECC requires $5.99\times$ more memory than the proposed error confinement approach.

6.3.5 Summary

In this work, a low cost error confinement technique is proposed which exploits the statistical characteristics of target applications and replaces any erroneous data with the best available estimate of that data. The architecture of a RISC processor with custom instructions supporting proposed approach is presented. The benchmarking

Figure 6.28: Execution time, data memory usage for error confinement vs. ECC

Figure 6.29: Energy ratio between error confinement and ECC vs. image size

result shows that the proposed approach achieves far less performance and memory usage overhead than ECC based error detection and correction, while also consumes less energy as image size grows. Further application-level studies using the proposed methodology will be presented in the future.

Chapter 7

System-level Reliability Exploration

In this chapter two techniques to enhance reliability in system-level design are proposed. First, a system-level exploration framework is presented in Section 7.1 which supports integration of heterogeneous processing elements and topology exploration. A novel task mapping algorithm targeting reliability is proposed and demonstrated on the platform. Second, an approach for reliable network design is illustrated in Section 7.2 based on the graph theoretical problem of Node Fault Tolerance.

7.1 System-level Reliability Exploration Framework

As task complexity increases, Multi-Processor System-on-Chip (MPSoC) becomes the state-of-the-art architecture for high performance and low power applications. System-level modelling techniques for MPSoC such as Transaction Level Modelling (TLM) using SystemC language are proposed due to their fast simulation speed and the ability to model systems with large number of processing units. Consequently, system-level reliability techniques which are compatible with architecture-level techniques gain their importance. To explore reliability in system-level design, efficient supports in tools, platforms and task mapping algorithms are essential.

System-level design tightly couples with task mapping techniques on MPSoC, which have been intensively investigated in recent past. A detailed survey on MPSoC task mapping can be found in [173]. With regard to the techniques improving device lifetime, [44] discusses approaches for addressing the lifetime optimization in terms of Mean-Time-To-Failure (MTTF). Coskun *et al* [39] presented a temperature-aware mapping that leads to increased lifetime. A wear-based heuristic is proposed in [78] to improve the system lifetime.

On the other hand, several papers target reliable mapping in presence of transient faults. In [106] the authors propose a remapping technique aimed towards determining task migrations with the minimum cost while minimizing the throughput degradation. In [166] a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems is presented. [47] evaluates several remapping algorithms for single fault scenarios by using Integer Linear Programming (ILP) under faulty core constraints. Several proposed heuristics also perform optimization to minimize communication traffic and total execution time.

Though reliability is treated by several research works for efficient task management, the proposed mapping techniques have not yet considered the intrinsic differences of reliability levels among different processing units. This is presumably due to the lack of system-level reliability exploration frameworks. The ERSA architecture [72]

addresses this issue by adopting one Super Reliable Core (SRC) and multiple Relaxed Reliable Cores (RRCs) and manages the probabilistic applications according to the vulnerability of the cores. Application-level asymmetric reliability requirements have been considered during task mapping. However, no generic task mapping algorithms jointly consider reliability levels of task and core have been proposed.

Contribution In this work, a heterogeneous multiprocessor platform consisting of processor IPs and customized modules for executing Kahn Process Network (KPN)-like [94] streaming applications is introduced. The processing elements and communication channels are equipped with fault injection properties proposed in Section 4.2. Executing on the centralized task manager, a novel firmware initializes user-defined KPN task graph and dynamically updates system interconnect topology. The task-mapping algorithm can be easily integrated through function interface in the firmware, thereby scheduling KPN applications accordingly. The mapping technique is further investigated in the presence of various reliability requirements among KPN tasks and different levels of reliability among heterogeneous processing elements. A combined task/core-reliability-aware task mapping heuristic is then presented.

7.1.1 Platform and Task Manager Firmware

In this section, the reliability exploration platform and task management methodologies are introduced. Figure 7.1 illustrates an exemplary heterogeneous MPSoC platform with a mapping example of KPN application. It is noted that KPN nodes have different reliability levels due to the application properties, which can be defined by software developer. For instance, higher reliability levels can be assigned to node P3 and P6 in Figure 7.1 due to higher degree of edges. From the architecture side, the ability to integrate customized processor helps improving core-level reliability. In Figure 7.1 the PD_RISC processors are protected with architecture-level fault tolerance features such as Error Correction Code (ECC) and Triple Modular Redundancy (TMR). During task mapping, task with high reliability level are preferred to be mapped on more reliable cores. To realize initial task mapping and run-time remapping, the run-time manager core, which is protected by both ECC and TMR, executes a firmware for task scheduling and monitoring under fault injection. The firmware is novel in the sense that it supports arbitrary platform topology and user defined task graphs through its API. Timer on individual processor informs the manager core whether the monitored processor is in unresponsive state and requires to be reset. The shared memory implements channels containing data tokens for communication between processors with synchronization features. Several novel features of the platform are presented in the following.

7.1.1.1 Customized processor integration

Taking advantage of Synopsys Processor Designer [182], customized processor in both RTL and SystemC package can be automatically generated from high-level descrip-

Figure 7.1: KPN tasks mapping to MPSoC considering node reliability level

tions. The PD_RISC core used as run-time manager and reliable processing elements (PEs) is a mixed 16 and 32 bits instruction set processor with 6 pipeline stages. Reliability extensions are implemented via additional LISA operations and resources. The processor-bus interface can be chosen among TLM 2.0 and AHB types depending on the applied bus system. Fault injection technique in Section 4.1 is applied for individual processor.

7.1.1.2 Run-time manager firmware

The extensibility of MPSoC platform requires support from the run-time manager for a dynamic platform topology specification, which considers not only system interconnects but also core reliability indexes due to intrinsic differences of fault tolerant abilities among heterogeneous cores. Both KPN application and platform topology are defined by the APIs shown in Figure 7.2.

Application graph Basic fields are used to describe the KPN task graph such as process ID and connecting processes. The firmware also maintains a look-up-table in the local memory of each PE for the function definition corresponding to the process ID. For reliability-directed mapping, user can provide reliability level for each process manually. A successful task mapping assigns PE IDs to all processes.

Platform topology Specific fields are required to represent the platform topology for each PE such as neighbouring PE nodes and connecting channels. For instance, bus based platform in Figure 7.1 is configured as a processor network with full connection. Architectural reliability index for each core is defined according to the EMR metric

Figure 7.2: Data structures for platform initialization

in Section 4.1.1.4. Detailed EMR evaluation for heterogeneous processors is referred in [199]. Besides, fault configuration is provided on each core for the purpose of fault injection.

Channels Channels implement not only inter-PE communication but also data synchronizations. Token type and buffer sizes are defined based on user inputs. Regarding the implementation, channels can be realized in different ways depending on the emulation platform. A NoC platform relates channels directly to its physical links. For a bus-based platform, channels are implemented as data structures in shared memory according to the topology to emulate generic styles of interconnects, which gets automatically analysed from topology graph. Fault injection in channels are implemented as bit manipulation in the data elements of channel structure, where a fault configuration file is provided for each channel. A specific *token state* field is used to pass the current task execution state to the following channels. It can be realized as an integer, whose value is incremented each time the start node (P1 in Figure 7.2) processes one token. When the same token is finished processing by the end node (P4), its value is updated in the shared memory. Such mechanism helps retrieving the processing state when run-time task remapping happens. After remapping, the start node can directly process the next token.

State transition Upon system initialization, the manager initializes KPN processes, topology and channels according to user-provided information while PEs wait for task

Figure 7.3: Run-time manager state transition

assignment. After a successful initial mapping, the PEs begin to perform individual tasks and token state begins to pass down the channels. The manager keeps checking the status of all PEs. The worst case is considered that unresponsive PE is not able to be restarted. Under such case whenever one PE is unresponsive, the platform topology is updated by removing the faulty PE and its edges from the topology graph. A task re-mapping phase then follows up. In case of mapping failure, the run-time manager terminates the system. A successful mapping will interrupt all PEs for task switching while current token state is retrieved to continue processing the erroneous token. The mapping algorithm can be realized as either complete run-time mapping or based on design-time analysis [173].

7.1.2 Core Reliability Aware Task Mapping

Focusing on core reliability-aware task mapping, the performance/power metrics among heterogeneous processors are currently disregarded. Besides, it is limited that only one task can be mapped to one PE, which implies a static global communication cost for a fixed KPN system. The focus of the remapping algorithm is to accept the core/task reliability constraint and generate a mapping with low overhead. A heuristic recursive mapping algorithm is developed in Algorithm 2. It maps tasks sequentially. Once a task is mapped successfully, the mapping of next dependent task in the task graph starts. Otherwise, the task will be mapped to other remaining processors. If such a task cannot be mapped to any remaining processor, the recursive algorithm returns and changes the previous task mapping. The algorithm stops when a successful mapping for all tasks are achieved.

Algorithm 2 Mapping task to platform recursively**INPUTS:** PE: Topology graph TA: Task graph**OUTPUT:** PE $\Leftrightarrow$ TA

```

1: function RUNMAP(PE, TA)
2:   sort_PE_node
3:   sort_TA_node
4:   status = recursiveMap(0)
5:   return status
6: end function
7:
8: function RECURSIVEMAP(task_id)
9:   if task_id == task_Count then
10:    return Success
11:   end if
12:   for pe_id = 1 to PE_Count do
13:     if mapT2P(task_id, pe_id) then
14:       binding(PE[pe_id], TA[task_id])
15:       if RECURSIVEMAP(task_id + 1) == Success then
16:         return Success
17:       else
18:         PE(pe_id)  $\rightarrow$  t_id = null
19:       end if
20:     end if
21:   end for
22:   return Fail
23: end function

```

▷ init recursive mapping
 ▷ last task has been mapped
 ▷ mapping plug-in
 ▷ recursive mapping success
 ▷ recursion fail, clear parent decision

Algorithm 3 shows the procedure which decides Task-PE mapping according to the constraints. Two constraints are presented while further ones considering other performance can be easily integrated.

Algorithm 3 Decision with edges and reliability constraints

```

1: function MAPT2P(task_id, pe_id)
2:   if PE(pe_id)  $\rightarrow$  Degree < TA(task_id)  $\rightarrow$  Degree then
3:     return Fail
4:   end if
5:   if PE(pe_id)  $\rightarrow$  relia_ind < TA(task_id)  $\rightarrow$  relia_lev then
6:     return Fail
7:   end if
8:   neighbors_ids = get_task_neighbors(task_id)
9:   for all neighbors_ids do
10:    pe_neb_id = TA(neighbors_ids)  $\rightarrow$  p_id
11:    if pe_neb_id != null then
12:      if is_pe_neighbors(pe_neb_id, pe_id) then
13:        else
14:          return Fail
15:        end if
16:      end if
17:    end for
18:    return Success
19: end function

```

▷ meet task edges constraint
 ▷ meet reliability requirement
 ▷ ensure task neighbors are topological neighbors

Task degree constraint The 1-to-1 mapping constraint implies a possible mapping only when the count of node edges in task graph is not larger than the count of PE edges in topology graph. Besides, connecting tasks in KPN graph should also be topological neighbours. The search procedure in Algorithm 2 starts by sorting both

processes and PEs in descending order of their degrees, which reduces the time for finding a possible mapping. During mapping, if the number of PE edges is smaller than required, the mapping fails. Otherwise, the task will check whether its dependent tasks, which have already been mapped, can reach it as topological neighbours.

Core reliability constraint A successful mapping ensures that the reliability indexes of all PEs are not less than the tasks' reliability level, which is considered every time before the mapping decision.

7.1.3 Experimental Results

In this section, several experimental studies are presented with the proposed techniques. Real-world KPN tasks are implemented on the customized MPSoC platform. Consequently, the effectiveness of run-time manager and core/task reliability-aware mapping is illustrated.

The efficacy of the mapping technique is explored with an audio processing application, shown as a KPN graph in Figure 7.4. The application is mapped onto a heterogeneous MPSoC platform with 16 PEs. The filter block task is assigned with a high reliability constraint according to its degree. To demonstrate the usage of proposed mapping algorithm, the platform consists PD_RISC processors with ECC protection on its program counter register (PC-register), which is labelled as 'H' while the rest ARM processors are labelled as 'L'.

Figure 7.4: KPN tasks mapping onto 16 PE platform

7.1.3.1 Algorithm constraints

Initially a fixed mapping as in 7.4a) is forced for all the tasks. Once single bit-flip is injected into the PC register, the ARM processor without ECC protection is likely to fall into unresponsive state which activates the run-time manager for task remapping. When only edge count constraint is applied, the tasks are mapped as in Figure 7.4b), where the filter task is still prone to the faults on an unreliable PE. However, a core-reliability-aware mapping schedules tasks as in Figure 7.4c), where further single bit-flip fault injection on the filter application does not hang the system due to the ECC protected program counter. Table 7.1 shows the required cycles of fault simulation to process 10 data tokens using different mapping algorithms. When the core reliability constraint is considered, an overhead of 1.2% is caused by task migration, while the system hangs when only edge count constraint is applied.

Mapping constraints	Cycles count w/o faults	Cycle count with faults	Cycle increased
edge count only	18,173k	Hang	Hang
edge count+core reliability	18,173k	18,387k	1.2%

Table 7.1: Mapping exploration with different algorithm constraints

7.1.3.2 Topology and PE types

Further mapping explorations with different topology and PE types are performed as in Figure 7.5. A platform with mesh topology suffers from 3 unresponsive PEs as in 7.5e). One extra high reliable core does not facilitate further remapping as shown in 7.5f). In the contrary, a topology with more links such as nearest neighbour (NN) realizes further mappings, where up to 5 unresponsive PEs are tolerant as in 7.5i). When further highly reliable core is deployed, remapping is still achieved with 6 hanging PEs as shown in 7.5k). No further mapping is possible with 7 hanging PEs.

Experiments are conducted where single bit-flip faults are injected to the PC registers of PEs as shown in Figure 7.5. Table 7.2 shows the required cycles to process 10 data tokens with regard to various topologies and PE types where up to 7 PEs become unresponsive during execution. It is shown that NN topology with 2 highly reliable PEs can tolerate up to 6 hanging PEs whereas Mesh suffers from 3 hanging PEs. The remapping task itself takes 143 kilo cycles on the supervisor core for 16-PE Mesh topology and 138 kilo cycles for the same using NN topology. In NN it is easier and faster to find a possible mapping according to the task degree constraint since all processor nodes have more edges than those in Mesh for executing the same KPN task. However, the increased amount of channels implies the trade-off between topology complexity and possibility of successful mapping. The overhead differences caused by varying number of reliable cores for the same topology and number of hanging PEs are minor since only difference of a few cycles is incurred during PE initialization.

The approach of design time analysis [166] is adopted and keep the mapping results in the local memory of task manager, so that mapping decisions can be directly

Figure 7.5: Mapping exploration for 7 KPN nodes

Hanging PE count	Cycle count (kcycles)					
	Mesh			NN		
	1H PE	2H PEs	+(%)	1H PE	2H PEs	+(%)
0	18,173	18,173	0	18,168	18,168	0
1	18,387	18,387	1.2	18,375	18,375	1.1
2	18,621	18,621	2.5	18,602	18,602	2.4
3	Hang	Hang	-	18,831	18,831	3.6
4	Hang	Hang	-	19,063	19,063	4.9
5	Hang	Hang	-	19,297	19,297	6.2
6	Hang	Hang	-	Hang	19,542	7.6
7	Hang	Hang	-	Hang	Hang	-

Table 7.2: Exploration with topology and PE types

retrieved with least computation overhead. Therefore, the task re-mapping overhead for NN topology compared with Mesh is increased less significantly, while the overhead differences caused by varying number of reliable cores for the same topology and number of hanging PEs are minor.

7.1.4 Summary

In this work a system-level reliability exploration framework is presented based on a commercial design flow. A mapping algorithm for process networks considering reliability level of individual tasks is illustrated. A heterogeneous MPSoC platform with user-defined architecture topology and its ability of integrating customized processors with reliability extension demonstrate the usability of the proposed mapping technique.

7.2 Reliable System-level Design using Node Fault Tolerance

System-level reliability exploration mainly addresses the problem of executing specific applications correctly on predefined multi-core platforms. Literature on such issues can be classified in two directions: reliable task mapping and reliable network design. While task mapping techniques have been heavily investigated in the past decade [173], design of reliable network topologies according to the application-level task pattern receives relatively less focus. The problem is partially addressed in the domain of Network-on-Chip (NoC), where different NoC topologies have been evaluated according to its robustness of successful message routing facing broken nodes/links and the ability to minimize routing delay. Representative NoC topologies are Mesh [191], Torus and Tree [178] which have been widely adopted in industry. Custom topologies such as de Bruijn graph [80] detour a faulty link while incurring less latency and energy consumption compared to Mesh and Torus. A survey on various NoC topologies can be referred in [61]. However, general network topology is either limited to ensure the reliable execution of specific applications or over designed with more links than necessary. An ad-hoc reliable topology customized for given task is highly demanded.

To program complex applications onto MPSoC, large applications are usually decomposed into smaller tasks, which runs on individual cores and communicates through on chip networks. Formally, such applications can be represented using graph structures with connected nodes such as Kahn Process Networks (KPN) [67]. Consequently, constructing reliable network topology can be derived from graph theoretical perspective, where the supergraph is isomorphic to the task graph when any of its nodes and connecting edges is removed. To find such supergraph with smallest amount of edges is the problem of Node Fault Tolerance (NFT). However, previous theory [77] handles only a subset of graphs, which limits its application for reliable ad-hoc network design.

Contribution In this work, the approach to construct NFT graphs is proposed by decomposing generic graph into basic subgraphs which can be individually handled using the theory in [77] and merged by manipulating the adjacency lists of subgraphs. The correctness of the proposed approach is further verified using exhaustive search

based task mapping algorithm and shows that a local optimal graph with least number of edges is achieved for the demonstrative task graph. Finally, the tasks are executed on the ad-hoc network topology and perform failure injection to indicate the efficiency of proposed approach using the framework in Section 7.1.

7.2.1 Node Fault Tolerance in Graph

In this section the system-level fault tolerance techniques are explored according to the graph-based theoretical models proposed by Harary and Hayes [77]. In the domain of Node Fault Tolerance (NFT), two graphs are involved in the discussion which are G and G^* . Graph G is always embedded in G^* even if the faults take out certain nodes and connecting edges out of G^* . Formally, G^* is the k -node-fault-tolerance graph of G , denoted as $k\text{-NFT}(G)$ when a maximal of k nodes failure can be tolerant in G^* . The G^* is not unique since many supergraphs which have k nodes more than G are $k\text{-NFT}(G)$. The problem of great interest is to find the G^* with minimal number of edges among all $k\text{-NFT}(G)$ graphs. Such G^* is named as *optimal* $k\text{-NFT}(G)$. Note that optimal $k\text{-NFT}(G)$ graphs are not unique. However, they all have same amount of nodes and edges. The problems of focus are formally stated as following:

- Given a graph G representing tasks and interconnects, construct supergraph G^* which is $k\text{-NFT}(G)$.
- Find the optimal G^* which has minimal number of edges among all G^* graphs.

For example, let a circle with 5 nodes indicated as C_5 in Figure 7.6(a) be the original task graph. A 1-NFT(C_5) graph is shown in Figure 7.6(b) consisting of a spare node (s). When the node denoted as (f) fails, C_5 is embedded in the supergraph G^* using the spare node, which is shown as the bold circle. Such G^* has in total 10 edges. Figure 7.6(c) gives the optimal 1-NFT(C_5) graph which has 9-edges. Figure 7.6(d) shows us the optimal 2-NFT(C_5) graph which has 14-edges.

Figure 7.6: (a) circle C_5 ; (b) non-optimal 1-NFT(C_5); (c) optimal 1-NFT(C_5); (d) optimal 2-NFT(C_5); [77] Copyright ©1996 John Wiley & Sons, Inc.

7.2.1.1 Optimal Node Fault Tolerance

Harary and Hayes [77] proposes the approach to achieve optimal k -NFT graph for circles and simple paths, which are illustrated in the following:

Harary-Hayes Theorem 1: *The two Hamiltonian-connected $(n+1)$ -node supergraphs depicted in Figure 7.7 (a) and (b) represent an optimal 1-NFT (C_n) for odd and even number of n .*

Figure 7.7: (a) 1-NFT(C_n) n odd; (b) 1-NFT(C_n) n even; (c) k -NFT(C_n) k even, n odd; (d) k -NFT(C_n) k odd, n even; [77] Copyright ©1996 John Wiley & Sons, Inc.

Let $k = 2h$ when k is even and $k = 2h + 1$ when k is odd. C_n^m represents the power graph obtained by adding edges to each node i of C_n which connects i to all nodes at distance m or less.

Harary-Hayes Theorem 2: *When k is even, the power graph C_{h+k}^{h+1} shown in Figure 7.7 (c) is an optimal k -NFT (C_n). When k is odd, the graph obtained by adding $\lfloor (n+k+1)/2 \rfloor$ bisector edges to C_{h+k}^{h+1} as shown in Figure 7.7 (d), is an optimal k -NFT (C_n).*

Harary-Hayes Theorem 3: *Let P_n be the simple path with n nodes. For any $k \geq 1$, k -NFT(P_n) = $(k-1)$ -NFT(C_{n+1}).*

Follow Theorem 3, the optimal k -NFT graph of paths can be constructed using Theorem 1 and 2.

7.2.2 Construct NFT for Arbitrary Graph

The proposed algorithm of exploring node fault tolerance for complicated graph is based on an extension of the Harary and Hayes' theorems. The heuristic algorithm make uses the concept of *divide and conquer* to obtain a near-optimal k -NFT supergraph. It works on arbitrary task graph consisting of elementary paths P_e and circles C_e which can be individually solved by Harary and Hayes' theorems. The elementary circles are the minimal circles which do not contain further circles inside.

Figure 7.8: The task graph G with nine nodes

Algorithm 4 shows the proposed divide and conquer approach to construct k -NFT graph for complicated task graph. After initializing the G^* as empty, the graph G is decomposed to individual elementary graphs G_e , whose k -NFT supergraph G_e^* is further constructed. After that the G^* is assigned by merging the G_e^* and current G^* .

Algorithm 4 Constructing k -NFT(G) for arbitrary graph G

INPUTS: G : Task graph; k : NFT level

OUTPUT: G^* : k -NFT graph G^*

```

1: function NFT( $G, k$ )
2:   initialize  $G^*$  as empty
3:   decompose  $G$  into elementary graph  $G_e$ 
4:   for each  $G_e$  do
5:     build  $G_e^* = k - \text{NFT}(G_e)$ 
6:      $G^* = \text{merge}(G_e^*, G^*)$ 
7:   end for
8:   return  $G^*$ 
9: end function

```

▷ according to Section 7.2.1

The construct of G_e^* from G_e follows the approach from Section 7.2.1. Here the routine of graph merging is described using the example shown in Figure 7.8 with 9 nodes, where 6 elementary circles exist.

Figure 7.9 represents the C_3 and C_4 graphs with their 1-NFT and 2-NFT supergraphs which can be constructed using Harary-Hayes Theorem 1 and 2. The spare nodes are shown in black color. Since no other elementary graphs are identified in 7.8, other constructions of NFT graphs are not listed. To represent the graph, the concept of adjacency list [38] is adopted, where the neighbour nodes of each one is represented as a list such as the one shown in Figure 7.10(a). For instance, Figure 7.10(b) represent the 1-NFT(C_4) graph in Figure 7.9(e).

Figure 7.10 shows three adjacency list representations of 1-NFT graphs for elementary circles and the list merging procedure represented as matrix operation. To merge two list A and B starting with the same node, it is only required to fill the absent element in list B into list A . The fault tolerant node 0 is the spare node for all

Figure 7.9: Optimal 1-NFT and 2-NFT graphs for C_3 and C_4

elementary circles which is put in the beginning of all lists. The merging is performed for all nodes in both graphs.

Follow the iterative procedure in Algorithm 4, all NFT graphs are merged which result in a final list representation of the supergraph G^* . The procedure is irrespective of the number of spare nodes k . Figure 7.11 illustrates the adjacency lists of the final 1-NFT and 2-NFT graphs for G in Figure 7.8. On the right side the corresponding network topologies are shown where the spare nodes are highlighted. For the 2-NFT case, node A and B are used to represent the spare nodes instead of node 0. The construct of final NFT graph does not depend on the order of circle selection and adjacency list transversal, which implies a robust algorithmic design.

It is obvious that a fully connected network topology G^* where each node connects to all other nodes can ensure both 1-NFT and 2-NFT of the original graph G . Compared with a full connection which incurs 45 edges for the network of 10 nodes and 55 edges for that of 11 nodes, the proposed approach results in 23 edges for 1-NFT(G) and 32 edges for 2-NFT(G). The saving of edges is 48.9% and 41.8% respectively.

7.2.3 Verify NFT Graphs using Task Mapping

The algorithm proposed in Section 7.2.2 analytically construct a k -NFT graph G^* using divide-and-conquer approach. To verify the optimality and even further reduce the number of edges in G^* while ensure the NFT condition, Algorithm 5 is proposed

$$\begin{aligned}
G_{e1}^* &= \begin{bmatrix} 0 & 1 & 2 & 4 & 6 \\ 1 & 0 & 2 & 6 & - \\ 2 & 0 & 1 & 4 & - \\ 4 & 0 & 2 & 6 & - \\ 6 & 0 & 1 & 4 & - \end{bmatrix} & G_{e2}^* &= \begin{bmatrix} 0 & 1 & 5 & 6 & 7 \\ 1 & 0 & 5 & 6 & - \\ 5 & 0 & 1 & 7 & - \\ 6 & 0 & 1 & 7 & - \\ 7 & 0 & 5 & 6 & - \end{bmatrix} & G_{e3}^* &= \begin{bmatrix} 0 & 6 & 7 & 9 \\ 6 & 0 & 7 & 9 \\ 7 & 0 & 6 & 9 \\ 9 & 0 & 6 & 7 \end{bmatrix} \\
& \text{(a)} & \text{(b)} & \text{(c)} \\
G^* &= \begin{bmatrix} 0 & 1 & 2 & 4 & 6 \\ 1 & 0 & 2 & 6 & - \\ 2 & 0 & 1 & 4 & - \\ 4 & 0 & 2 & 6 & - \\ 6 & 0 & 1 & 4 & - \\ - & - & - & - & - \\ 0 & 1 & 5 & 6 & 7 \\ 1 & 0 & 5 & 6 & - \\ 5 & 0 & 1 & 7 & - \\ 6 & 0 & 1 & 7 & - \\ 7 & 0 & 5 & 6 & - \\ - & - & - & - & - \\ 0 & 6 & 9 & 7 & - \\ 6 & 0 & 7 & 9 & - \\ 7 & 0 & 6 & 9 & - \\ 9 & 0 & 6 & 7 & - \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 4 & 6 & 5 & 7 \\ 1 & 0 & 2 & 6 & 5 & - & - \\ 2 & 0 & 1 & 4 & - & - & - \\ 4 & 0 & 2 & 6 & - & - & - \\ 6 & 0 & 1 & 4 & 7 & - & - \\ 5 & 0 & 1 & 7 & - & - & - \\ 7 & 0 & 5 & 6 & - & - & - \\ - & - & - & - & - & - & - \\ 0 & 6 & 9 & 7 & - & - & - \\ 6 & 0 & 7 & 9 & - & - & - \\ 7 & 0 & 6 & 9 & - & - & - \\ 9 & 0 & 6 & 7 & - & - & - \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 4 & 6 & 5 & 7 & 9 \\ 1 & 0 & 2 & 6 & 5 & - & - & - \\ 2 & 0 & 1 & 4 & - & - & - & - \\ 4 & 0 & 2 & 6 & - & - & - & - \\ 6 & 0 & 1 & 4 & 7 & 9 & - & - \\ 5 & 0 & 1 & 7 & - & - & - & - \\ 7 & 0 & 5 & 6 & 9 & - & - & - \\ 9 & 0 & 6 & 7 & - & - & - & - \end{bmatrix} \\
& \text{(d)}
\end{aligned}$$

Figure 7.10: Merge of three 1-NFT graphs

Figure 7.11: Final 1-NFT(G) and 2-NFT(G)

which takes advantage of the recursive task mapping routines from Algorithm 2 and 3 in Section 7.1.2.

Algorithm 5 Reduce number of edges in graph G

INPUTS: G : Task graph; G^* : Output task graph from Algorithm 4

OUTPUT: G_{opt}^* : Local optimal k -NFT graph

```

1: function NFT_REDUCTION( $G, G^*$ )
2:    $status = \text{NFT\_VERIFY}(G, G^*, k)$ 
3:   if  $status == \text{Success}$  then
4:      $G_{old} = G^*$ 
5:     for each edge  $e$  in  $G^*$  do
6:        $G^* = G^* - e$  ▷ remove edge  $e$  from  $G^*$ 
7:        $G_{new} = \text{NFT\_REDUCTION}(G, G^*)$ 
8:       if  $|E(G_{new})| < |E(G_{old})|$  then ▷ compare count of edges from  $G_{new}$  and  $G_{old}$ 
9:          $G_{old} = G_{new}$ 
10:      end if
11:    end for
12:     $G^* = G_{old}$ 
13:  end if
14:  return  $G^*$ 
15: end function
16:
17: function NFT_VERIFY( $G, G^*, k = 1$ ) ▷ Assume  $k = 1$  for simplicity
18:  for each node  $n$  in  $G^*$  do
19:     $G_{chk} = G^* - n$  ▷ remove node  $n$  from  $G^*$ 
20:     $status = \text{RUNMAP}(G_{chk}, G)$  ▷ Algorithm 2 and 3 in Section 7.1.2
21:    if  $status == \text{Fail}$  then return  $\text{Fail}$ 
22:  end if
23: end for
24: return  $\text{Success}$ 
25: end function

```

Algorithm 5 consists of two routines, the `NFT_REDUCTION` which reduces number of edges and `NFT_VERIFY` which verifies the NFT condition for given G and G^* .

- **NFT_VERIFY:** tries to remove k nodes and their connecting edges from G^* and applies the task mapping routine to map G onto the new G^* . The new G^* is $k - \text{NFT}(G)$ if any selection of k nodes and connecting edges in G^* are removable while at least one mapping is found.
- **NFT_REDUCTION:** provided that G^* is $k - \text{NFT}(G)$, this routine tries to remove one edge per iteration out of all edges and check if the resulted graph is still edge reducible. The recursive function ensures a local optimal G_{opt}^* according to the initial graph G^* .

In contrast to the k -NFT graph G^* resulted from Algorithm 4, Algorithm 5 can also take a fully connected graph G^* so that the global optimal graph G_{opt}^* will be achieved. However, such exhaustive search can take recursively up to 2^{45} choices to remove edges in `NFT_REDUCTION` for the fully connected graph with 10 nodes, which resulted in unacceptable time and complexity even for constructing an 1-NFT graph. By providing G^* from the analytical approach, the maximal number of choices reduces to 2^{23} since only 23 edges remains in $1 - \text{NFT}(G)$ graph from Figure 7.11. In contrast to the function of edge reduction, the exhaustive search based graph verifica-

Faulty Units	Task index								
	T0	T1	T2	T3	T4	T5	T6	T7	T8
P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
P1	P0	P2	P3	P4	P5	P6	P7	P8	P9
P2	P1	P0	P3	P4	P5	P6	P7	P8	P9
P3	P2	P4	P6	P8	P1	P0	P5	P7	P9
P4	P2	P1	P6	P5	P3	P0	P8	P7	P9
P5	P1	P2	P3	P4	P6	P0	P7	P8	P9
P6	P1	P2	P3	P4	P5	P0	P7	P8	P9
P7	P1	P6	P9	P4	P5	P2	P0	P8	P3
P8	P1	P2	P3	P4	P5	P6	P7	P0	P9
P9	P6	P4	P8	P2	P7	P1	P5	P3	P0

Table 7.3: Task remapping for faulty PEs under 1-NFT topology

tion function achieves a time complexity of $O(n^k)$, which increases exponentially with the NFT requirement factor k .

The combination of the analytical approach and exhaustive search method is utilized to find a local minimal G_{opt}^* . It is interesting that no edge among the 23 edges in G^* is detected to be removable which still satisfies the 1-NFT condition. Consequently, the $1 - NFT(G)$ graph in Figure 7.11 represents the local optimal graph for task G . The conclusion that it is also the global optimal 1-NFT graph can not be reached due to the huge searching space and time using the exhaustive search algorithm.

7.2.4 Experiments for Node Fault Tolerance

To explore system-level reliability using node fault tolerance, the processor network is constructed according to Section 7.2.2 and map the task in Figure 7.8. Fault injection experiments are conducted on individual core to change it into the state of unresponsiveness. The firmware detects such situation and initiate the task mapping routine according to Section 7.1.1.2. Table 7.3 and 7.4 show the remapping schemes under 1-NFT and 2-NFT condition respectively using the supergraph in Figure 7.11. For the 1-NFT scheme, 1 faulty core is presented with the tasks and remapping cores. For the 2-NFT scheme, a pair of faulty cores are presented for the selected schemes. It is found that under all faulty cases, a successful task remapping scheme is found.

Figure 7.12(c) and (d) visualize the 1-NFT based task mapping where 1 out of 10 processing elements is faulty. The mapping reflects the schemes is Table 7.3.

Finally, tasks are assigned with real operators and execute them using the system-level reliability exploration framework in Section 7.1. The platform consists of one task manager ARM9 core and 11 ARM9 as processing elements. Figure 7.13(a) shows the directed task graph with extra input and output signals which match exactly the graph pattern in Figure 7.8. The corresponding operators for each task are shown in Figure 7.13(b). Only simple operators are applied to avoid the long simulation time for real world tasks. 100 data tokens are provided to task 4 as input and outputs are

Faulty Units	Task index									Faulty Units	Task index								
	T0	T1	T2	T3	T4	T5	T6	T7	T8		T0	T1	T2	T3	T4	T5	T6	T7	T8
P0,P1	P10	P4	P3	P9	P8	P7	P6	P5	P2	P3,P2	P7	P9	P10	P5	P6	P0	P8	P4	P1
P0,P2	P7	P5	P9	P3	P6	P1	P8	P4	P10	P3,P4	P2	P1	P5	P10	P6	P0	P8	P9	P7
P0,P3	P2	P5	P9	P4	P6	P1	P8	P10	P7	P3,P5	P10	P8	P7	P6	P4	P0	P9	P2	P1
P0,P4	P9	P10	P7	P8	P5	P1	P3	P6	P2	P3,P6	P4	P5	P9	P2	P10	P0	P8	P7	P1
P0,P5	P9	P10	P7	P8	P4	P1	P3	P6	P2	P3,P7	P2	P5	P4	P9	P6	P0	P8	P10	P1
P0,P6	P4	P10	P9	P8	P3	P1	P2	P7	P5	P3,P8	P7	P5	P9	P4	P6	P0	P2	P10	P1
P0,P7	P8	P10	P4	P9	P6	P1	P2	P5	P3	P3,P9	P2	P7	P5	P10	P6	P0	P8	P4	P1
P0,P8	P7	P9	P5	P10	P6	P1	P2	P4	P3	P3,P10	P5	P2	P7	P6	P4	P0	P9	P8	P1
P0,P9	P4	P10	P7	P8	P3	P1	P5	P6	P2	P4,P2	P9	P7	P10	P6	P5	P0	P3	P8	P1
P0,P10	P2	P3	P5	P4	P6	P1	P8	P9	P7	P4,P5	P6	P7	P8	P9	P2	P0	P3	P10	P1
P1,P2	P10	P4	P9	P3	P8	P0	P6	P5	P7	P4,P6	P5	P7	P9	P8	P3	P0	P2	P10	P1
P1,P3	P2	P5	P9	P4	P6	P0	P8	P10	P7	P4,P7	P10	P1	P3	P9	P8	P0	P6	P5	P2
P1,P4	P9	P10	P7	P8	P5	P0	P3	P6	P2	P4,P8	P6	P7	P5	P10	P2	P0	P3	P9	P1
P1,P5	P3	P2	P7	P6	P4	P0	P9	P8	P10	P4,P9	P2	P7	P10	P6	P3	P0	P5	P8	P1
P1,P6	P4	P10	P9	P8	P3	P0	P2	P7	P5	P4,P10	P3	P2	P7	P6	P5	P0	P9	P8	P1
P1,P7	P10	P4	P3	P9	P8	P0	P6	P5	P2	P5,P6	P4	P10	P9	P8	P3	P0	P2	P7	P1
P1,P8	P6	P7	P9	P10	P2	P0	P3	P4	P5	P5,P7	P6	P8	P1	P10	P2	P0	P3	P9	P4
P1,P9	P10	P4	P5	P3	P8	P0	P6	P2	P7	P5,P8	P6	P7	P9	P10	P2	P0	P3	P4	P1
P1,P10	P2	P3	P5	P4	P6	P0	P8	P9	P7	P5,P9	P2	P7	P10	P6	P3	P0	P4	P8	P1

Table 7.4: Selected task remapping for faulty PEs under 2-NFT topology

read out from task 1. Each data token consists of the flag section and data section. The flag section is used by the firmware on the PE to synchronize data execution.

An infinite loop is used on the faulty PE to imitate the state of unresponsiveness. The manager core detects such unresponsiveness using the down count timer per PE and initiate the task remapping routine. The core reset is deactivate to ensure the NFT style task mapping. Task mapping table is stored in Table 7.3 and 7.4 as Look-up-Table so that new mapping patterns can be retrieved in small amount of time. Figure 7.13(c) and (d) represent the execution cycles to finish processing 100 data tokens under faulty conditions. Compared with golden simulation, each time of remapping consumes around 1,000 clock cycles for both 1-NFT and 2-NFT cases. With the approach proposed in this work, the processor network survives fault injections and produce correct data outputs, regardless of which cores are faulty.

7.2.5 Summary

In this work, task pattern in Figure 7.8 is applied as example to show the divide-and-conquer technique of NFT graph construction and its verification. Following the same approach a ad-hoc NFT network can be analytically constructed for arbitrary graphs consisting of elementary circle and paths. The graph verification function achieves a timing complexity of $O(n^k)$, which applies the exhaustive search method.

Figure 7.12: NFT mapping schemes with one faulty core

(a) Directed process network

Node	Operator
T0	$in1 \ll 2$
T1	$OUT = in1 * in2 * in3$
T2	$in1 \& in2$
T3	$in1 \wedge in2$
T4	$IN \gg 2$
T5	$in1 in2$
T6	$in1 \bmod 2^5$
T7	$in1 * 8$
T8	$in1 + in2$

(b) Exemplary operator for each process node

Faulty PE index	Cycle count to process 100 data tokens
P0	15,123
P1	15,201
P2	15,110
P3	15,324
P4	15,100
P5	15,157
P6	15,121
P7	15,159
P8	15,113
P9	15,005
No fault	14,115

(c) Tasks execution with 1-NFT

Faulty PE index	Cycle count to process 100 data tokens
P0,P1	15,969
P2,P3	15,911
P4,P5	16,072
P6,P7	16,118
P8,P9	16,094
No faults	14,115

(d) Tasks execution with 2-NFT

Figure 7.13: Task execution time under 1-NFT and 2-NFT

Chapter 8

Conclusion and Outlook

8.1 Conclusion

Continuous technology scaling in semiconductor industry forces reliability as a serious design concern in the era of nanoscale computing. Traditional low level reliability estimation and fault tolerant techniques neither address the huge design complexity of modern system-on-chip nor consider architectural and system-level error masking properties. According to International Technology Roadmap for Semiconductors (ITRS), reliability and resilience across all design layers constitute a long-term grand challenge.

To enable cross-layer exploration of reliability against other performance constraints, it is essential to accurately model the errors in nanoscale technology and develop a smooth tool-flow at high-level design layers to estimate error effects, which assists the development of high-level fault-tolerant techniques. In this dissertation, several challenges are tackled for developing an high-level reliability estimation and exploration framework, which are identified as following.

- **High-level Fault Injection and Simulation**

A high-level fault injection tool is constructed for generic cycle-accurate architecture models which has been integrated into commercial processor design framework. Two modes of fault injection are supported which are the user configurable mode and timing error mode. The fault injector is further extended for system-level modules. A power/thermal/logic delay co-simulation framework is presented as a case study for integrating fault injection with simulation of physical properties.

- **High-level Reliability Estimation**

Three techniques are proposed to estimate the reliability for computing elements. The analytical method utilizes Directed Acyclic Graph to calculate vulnerability and error masking capability of individual logic blocks. Instruction and application-level error probabilities are further calculated through the graph structure. A formal algorithmic technique is introduced to predict error effects by tracking error propagation in a graph network representing dynamic processor behavior. The traditional design diversity metric is extended to quantify the robustness of major computing elements using Conflict Multiplex Graph.

- **Architectural Reliability Exploration**

Three architectural fault-tolerant techniques are proposed. Opportunistic redundancy presents a passive error detection approach for algorithmic units by

re-executing the instruction only if there exists underutilized resources, which incurs very small performance penalty. Asymmetric redundancy introduces an unequal error protection technique for storage elements based on criticality analysis of instruction and data. Error confinement exploits the statistical characteristics of target application and replaces any erroneous result with the best available estimate rather than correcting every single error. All techniques are demonstrated on embedded processors with customized architecture extension.

- **System-level Reliability Exploration**

System-level fault tolerant techniques are presented which focus on reliability-aware task mapping and reliable network design. A heuristic task mapping algorithm which jointly considers task reliability requirement and core reliability level is demonstrated on a heterogeneous multiprocessor platform with customized firmware layer for fault injection, system topology and task management. A theoretical approach to construct ad-hoc fault tolerant network for arbitrary task graph with optimal amount of connecting edges is presented and verified using exhaustive search based algorithm.

8.2 Outlook

The techniques proposed in this dissertation assist further research in high-level reliability estimation and exploration. Several future research directions are outlined in the following.

- **System-level Impact of Physical Errors**

A wide range of physically characterized fault models can be integrated into the proposed fault simulator, which is based on instruction-set simulation and achieves orders of speed-up compared with RTL and Gate-level simulations. This facilitates the investigation of fault effects on application level. For instance, to which extent can the errors imposed by dynamic frequency scaling be tolerated for machine learning algorithm? What is the system-level impact of voltage variation? How does the ageing of gates caused by NBTI reduce the image quality under processing? To solve such question the fault simulation based on realistic physical models needs to be performed with acceptable simulation speed.

- **High-level Design and Synthesis for Reliability**

The reliability estimation techniques can be fluently integrated into a high-level architecture design and synthesis framework. For instance, high-level synthesizer can select hardware modules with sufficient reliability level according to user's constraints. Designer can fast estimate the reliability of individual modules through fault injection and PeMM, while exploring the trade-off between reliability and area using design diversity. Processor designer can always update the instruction error properties using the analytical technique for any custom logic and instruction.

- **Software and Compiler Techniques for Fault Tolerance**

The architectural fault tolerance techniques can be directly involved with software and compiler optimizations. Opportunistic redundancy indicates the trade-off between code length with spatial redundancy, which can be of particular interests in the code generation phase for parallel structures such as VLIW. Error confinement gives the possibility of low cost protection according to the importance of data word, which can be customized by software designer. System-level designer can guide multiprocessor task mapping according to the reliability requirement and robustness of individual cores.

- **Novel Techniques in Approximate Computing**

Many proposals in this dissertation are inspired by approximate computing. Asymmetric redundancy protects most important data with highest redundancy level. Error confinement takes advantage of application characteristics to confine the error using statistical mean value. Such techniques save huge processing power compared with traditional error detection and correction techniques such as ECC and check-pointing. Since approximate computing is a rising topic which are still mainly investigated in low-levels, it is believed that the proposed high-level framework will definitely assist researchers for further development.

- **Fault Tolerance in Network Design**

The proposed divide-and-conquer approach of reliable network design based on Node Fault Tolerance can find its usage in the domain of manycore and super computing. Instead of aggressive task migration in a complex network of processing elements which imposes large processing power, the ad-hoc NFT network with relative smaller amount of edges can guarantee the functional state facing pre-defined amount of failure cores. A partner idea focusing on Edge Fault Tolerance is also worth investigating for arbitrary process networks.

Glossary

Acronyms

ACE	Architecturally Correct Execution
ADL	Architecture Description Language
AER	Application Error Rate
AES	Advance Encryption Standard
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Integrated Processor
AVF	Architecture Vulnerability Factor
BDD	Binary Decision Diagram
BER	Bit Error Rate
CCS	Concurrent and Comparative Simulation
CGRA	Coarse Grained Reconfigurable Architecture
CM	Code Modification
CMF	Common Mode Failure
CMG	Conflict Multiplex Graph
CMOS	Complementary Metal Oxide Semiconductor
CRT	Chip-level Redundant Threading
DAG	Directed Acyclic Graph
DCH	Divide and Conquer Hamming
DCT	Discrete Cosine Transformation
DFS	Dynamic Frequency Scaling
DMR	Dual-modular Redundancy
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
DTA	Dynamic Timing Analysis
DUE	Detected Unrecoverable Error
ECC	Error Correcting Code
EFT	Edge Fault Tolerance

EM	Electromigration
EMR	Error Manifestation Rate
ESL	Electronic System Level
FI	Fault Injection
FIT	Failure-in-Time
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HCI	Hot Carrier Injection
IDCT	Inverse Discrete Cosine Transformation
IER	Instruction Error Rate
IIE	Inter-Instruction Effect
ILP	Integer Linear Programming
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ITD	Inverted Temperature Dependence
ITM	Ideal Transfer Matrix
ITRS	International Technology Roadmap for Semiconductors
IVF	Instruction Vulnerability Factor
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
LISA	Language for Instruction Set Architecture
LLVM	Low Level Virtual Machine
LUT	Look-Up Table
MBU	Multiple Bit Upset
MCU	Multiple Cell Upset
MPSoC	Multi-Processor System-on-Chip
MSE	Mean Square Error
MTTF	Mean-Time-to-Failure
NBTI	Negative Bias Temperature Instability
NFT	Node Fault Tolerance
NN	Nearest Neighbour
NoC	Network-on-Chip
NOP	No Operation
OSIP	Operating System Application Specific Instruction-set Processors
PC-register	Program Counter Register
PE	Processing Element
PeMM	Probabilistic error Masking Matrix

PSNR	Peak Signal to Noise Ratio
PTM	Probabilistic Transfer Matrix
RC-circuit	Resistor–Capacitor circuit
RISC	Reduced Instruction Set Computer
RMS	recognition, mining and synthesis
RMT	Redundant Multithreading
RRC	Relaxed Reliable Core
RTL	Register Transfer Level
SAT	Boolean Satisfiability
SBU	Single Bit Upset
SC	Simulator Commands
SDC	Silent Data Corruption
SECDED	Single Error Correction Double Error Detection
SER	Soft Error Rate
SET	Single Event Transient
SEU	Single Event Upset
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory
SRC	Super Reliable Core
SRT	Simultaneous and Redundantly Threaded
STA	Static Timing Analysis
TLM	Transaction Level Modeling
TMR	Triple-modular Redundancy
VCD	Value Change Dump
VLIW	Very Long Instruction Word
VPI	Verilog Programming Interface
ZOL	Zero Overhead Loop
ZTC	Zero-Temperature Coefficient

List of Figures

1.1	Overall flow of high-level reliability estimation and exploration	2
2.1	SER scale trend for SRAM and DRAM [176] Copyright ©2010 IEEE . . .	9
2.2	SER scale trend for combinatorial logic [171] Copyright ©2002 IEEE . .	10
4.1	LISA-based fault injection and evaluation flow	33
4.2	Logic fault injection through disturbance signals	34
4.3	Graphical user interface for fault configuration and evaluation	35
4.4	Simulator extension for injection of delay faults	38
4.5	Exemplary EMR with increasing duration of fault (RISC)	40
4.6	Exemplary EMR with increasing number of fault (RISC)	41
4.7	Exemplary EMR with increasing duration of fault (VLIW)	42
4.8	System-level fault injection on virtual prototype	45
4.9	H.264 decoder with fault injection	47
4.10	Median filter: original and filtered image	48
4.11	Median filter: reliability exploration	48
4.12	LISA-based power modelling and simulation flow	51
4.13	Hierarchical representation of RISC processor architecture	52
4.14	Unit-level power model	53
4.15	Processor resource table for HDL generation	54
4.16	Separate modes for power models	55
4.17	Instruction-level power for RISC processor	57
4.18	Application profiling and average power	58
4.19	Instantaneous power for selected applications	59
4.20	Floorplan information for input of HotSpot framework	60
4.21	Instantaneous temperature generated by HotSpot	61
4.22	Thermal-aware fault injection	64
4.23	Critical paths and transverse blocks	65
4.24	Delay variation function under several conditions	66
4.25	Runtime delay of critical path for BCH application	66
4.26	Automation flow of power/thermal/logic delay co-simulation	67
5.1	ADL driven reliability estimation flow	72
5.2	Data flow graph for ALU instruction	73
5.3	Operation graph for all instructions in RISC processor	75

5.4	Faults in logic circuits	80
5.5	Probabilistic error Masking Matrix (PeMM)	81
5.6	Logic blocks involved for ALU instruction	82
5.7	Decomposition of large logic block using PeMM	83
5.8	Control flow handling for PeMM	84
5.9	Byte-level PeMM	86
5.10	Nibble-level PeMM	87
5.11	Error tracking and prediction framework	88
5.12	Comparison of error prediction accuracy for different modes of PeMM against fault injection [42]	91
5.13	Run-time among different PeMM modes	93
5.14	Error prediction for median filter application	94
5.15	Duplex and multiplex redundant systems	95
5.16	Implementation for Full Adder (FA) and Full Subtractor (FS)	97
5.17	Directed acyclic graph with ISA coding for ADL model	98
5.18	Conflict graph for selected operators in Figure 5.17	99
5.19	Directed acyclic graph and conflict multiplex graph	100
5.20	Conflict multiplex graph for TMR Architecture	101
5.21	Conflict multiplex graph for URISC Architecture	102
5.22	Conflict multiplex graph for VLIW Architecture	102
5.23	Conflict multiplex graph for CGRA Architecture	103
5.24	Design diversity of architecture variants	104
5.25	Application-level design diversity for PD_RISC processor	105
5.26	Mean-time-to-failure of architecture variants	107
6.1	Directed acyclic graph of embedded RISC processor	111
6.2	Average instruction distribution for MiBench	112
6.3	Protection features and policies	113
6.4	Flow of protection unit	114
6.5	PD_RISC_32p6 processor with protection modification	115
6.6	VLIW architecture with protection modification	117
6.7	VLIW control register	118
6.8	Instruction coverages and performance degradation	119
6.9	EMR with increased count of faults for RISC/VLIW processor	119
6.10	Effects of C compiler optimization levels on EMR for passive mode . . .	120
6.11	Asymmetric encoding and decoding	124
6.12	ECC encoding and decoding	127
6.13	EMR with different protection modes (Sieve application on RISC pro- cessor)	128
6.14	Static instruction criticality assignment	129
6.15	FSM for dynamic, asymmetric reliability	130
6.16	Comparing static and dynamic protection	131
6.17	ECC in VLIW slots	132
6.18	EMR for different protection modes: VLIW	133

6.19	Bit-wise asymmetric encoding	134
6.20	Comparing symmetric and asymmetric bit-wise protection	134
6.21	Microarchitecture of RISC processor with enhancements for statistical based error confinement	137
6.22	Introduced modules and their functionalities	138
6.23	Subsystem in JPEG application	139
6.24	Reference matrix for DCT and Quantization Coefficients	139
6.25	Programming example with custom instructions for DCT	140
6.26	Output images under different schemes of error injection	142
6.27	PSNR under no protection, proposed scheme and ECC	143
6.28	Execution time, data memory usage for error confinement vs. ECC	145
6.29	Energy ratio between error confinement and ECC vs. image size	145
7.1	KPN tasks mapping to MPSoC considering node reliability level	149
7.2	Data structures for platform initialization	150
7.3	Run-time manager state transition	151
7.4	KPN tasks mapping onto 16 PE platform	153
7.5	Mapping exploration for 7 KPN nodes	155
7.6	(a) circle C_5 ; (b) non-optimal 1-NFT(C_5); (c) optimal 1-NFT(C_5); (d) op- timal 2-NFT(C_5); [77] Copyright ©1996 John Wiley & Sons, Inc.	157
7.7	(a) 1-NFT(C_n) n odd; (b) 1-NFT(C_n) n even; (C) k -NFT(C_n) k even, n odd; (d) k -NFT(C_n) k odd, n even; [77] Copyright ©1996 John Wiley & Sons, Inc.	158
7.8	The task graph G with nine nodes	159
7.9	Optimal 1-NFT and 2-NFT graphs for C_3 and C_4	160
7.10	Merge of three 1-NFT graphs	161
7.11	Final 1-NFT(G) and 2-NFT(G)	161
7.12	NFT mapping schemes with one faulty core	165
7.13	Task execution time under 1-NFT and 2-NFT	166

List of Tables

4.1	Currently implemented fault types	31
4.2	Fault properties in configuration file	36
4.3	Comparison of fault simulation speed	42
4.4	Comparison of Synthesis Result	44
4.5	Power estimation accuracy for each instruction group	56
4.6	Power estimation for custom instruction	58
4.7	Temperature and power of LT_RISC at different frequencies running BCH application	62
4.8	Temperature of LT_RISC running BCH application using different floor- plans	62
4.9	Temperature of LT_RISC at 500MHz for different applications	63
4.10	Time and accuracy of power characterization for testbench groups . . .	69
4.11	Runtime overhead for different simulation modes	69
5.1	Instruction-level reliability estimation	76
5.2	Reliability estimation for selected applications	78
5.3	Examples of PeMM elements with byte-level granularity	85
5.4	Speed and accuracy using proposed framework	91
5.5	Processing time for automated PeMM preparation	92
5.6	Timing overhead analysis against architecture simulator	92
5.7	Design diversity for duplex pairs in Figure 5.16	97
5.8	Duplex pairs for EX pipeline stage in Figure 5.19	100
5.9	Architecture variants of design diversity evaluation	104
5.10	Failure rate estimation for four operators	106
6.1	Handling methods of different instruction types	115
6.2	ALU control register	117
6.3	Design overheads for proposed architectures	121
6.4	DCH schemes from different message partitioning	125
6.5	Performances for different protection modes	128
6.6	Reliability vs. power/area trade-off	131
6.7	Application runtime for various VLIW protection modes	132
6.8	Results for the proposed architecture extensions compared to the refer- ence unprotected processor	144

7.1	Mapping exploration with different algorithm constraints	154
7.2	Exploration with topology and PE types	155
7.3	Task remapping for faulty PEs under 1-NFT topology	163
7.4	Selected task remapping for faulty PEs under 2-NFT topology	164

Bibliography

- [1] Synopsys platform architect introduction. Accessed: 2013-07-21. [Online]. Available: <http://www.synopsys.com/Systems/ArchitectureDesign/Pages/PlatformArchitect.aspx>
- [2] A. Chattopadhyay, H. Meyr and R. Leupers, *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*. Morgan Kaufmann, jun 2008, ch. 5, pp. 95–130.
- [3] A. Kahng, S. Kang, R. Kumar, and J. Sartori, “Designing Processors from the Ground up to Allow Voltage/Reliability Tradeoffs,” in *HPCA*, 2010.
- [4] M. A. Aguirre, J. N. Tombs, V. Baena, F. Muñoz-Chavero, A. Torralba, A. Fernández-León, and F. Tortosa, “Ft-unshades: A new system for seu injection, analysis and diagnostics over post synthesis netlist,” *Proc. NASA Military and Aerospace Programmable Logic Devices, MAPLD*, Washington, DC, 2005.
- [5] S. B. Akers and B. Krishnamurthy, “On group graphs and their fault tolerance,” *Computers, IEEE Transactions on*, vol. 100, no. 7, pp. 885–888, 1987.
- [6] M. A. Alam and S. Mahapatra, “A comprehensive model of pmos nbt degradation,” *Microelectronics Reliability*, vol. 45, no. 1, pp. 71–81, 2005.
- [7] M. Alderighi, F. Casini, S. D’Angelo, M. Mancini, D. M. Codinachs, S. Pastore, C. Poivey, G. R. Sechi, G. Sorrenti, and R. Weigand, “Experimental validation of fault injection analyses by the flipper tool,” *IEEE Transactions on Nuclear Science*, vol. 4, no. 57, pp. 2129–2134, 2010.
- [8] D. Alexandrescu, E. Costenaro, and M. Nicolaidis, “A practical approach to single event transients analysis for highly complex designs,” in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2011 IEEE International Symposium on. IEEE, 2011, pp. 155–163.
- [9] J. Arlat, Y. Crouzet, and J.-C. Laprie, “Fault injection for dependability validation of fault-tolerant computing systems.” in *FTCS*, 1989, pp. 348–355.
- [10] <http://www.arm.com/products/processors/classic/arm9/index.php>, ARM.
- [11] S. I. Association *et al.*, “International technology roadmap for semiconductors (itrs), 2003 edition,” *Hsinchu, Taiwan, Dec*, 2003.
- [12] T. Austin *et al.*, “DIVA: A reliable substrate for deep submicron microarchitecture design,” in *International Symposium on Microarchitecture*, 1999, pp. 196–207.

- [13] A. Avizienis and J. P. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.
- [14] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during execution," in *Proc. IEEE COMPSAC*, vol. 77, 1977, pp. 149–155.
- [15] B. Nicolescu and R. Velazco, "Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003.
- [16] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.
- [17] N. Banerjee, G. Karakonstantis, and K. Roy, "Process variation tolerant low power dct architecture," in *Proceedings of the conference on Design, automation and test in Europe*. EDA Consortium, 2007, pp. 630–635.
- [18] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A prototype of a vhdl-based fault injection tool: description and application," *Journal of Systems Architecture*, vol. 47, no. 10, pp. 847–867, 2002.
- [19] G. Beltrame, L. Fossati, and D. Sciuto, "Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1857–1869, 2009.
- [20] L. Berrojo, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, C. Lopez *et al.*, "New techniques for speeding-up fault-injection campaigns," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*. IEEE, 2002, pp. 847–852.
- [21] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula, "Predictive modeling of the nbti effect for reliable design," in *Custom Integrated Circuits Conference, 2006. CICC'06. IEEE*. IEEE, 2006, pp. 189–192.
- [22] A. BIOS, "kernel developer's guide for amd npt family 0fh processors, jul. 2007," URL http://support.amd.com/us/Processor_TechDocs/32559.pdf.
- [23] A. Biswas, P. Racunas, J. S. Emer, and S. S. Mukherjee, "Computing Accurate AVFs using ACE Analysis on Performance Models: A Rebuttal," *Computer Architecture Letters*, vol. 7, no. 1, pp. 21–24, 2007.
- [24] H. Blume, D. Becker, M. Botteck, J. Brakensiek, and T. Noll, "Hybrid functional and instruction level power modeling for embedded processors," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 216–226, 2006.
- [25] S. Borkar, "The exascale challenge," *Proc. VLSI-DAT*, 2010.
- [26] J. Boué, P. Pétilion, and Y. Crouzet, "Mefisto-l: a vhdl-based fault injection tool for the experimental assessment of fault tolerance," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. IEEE, 1998, pp. 168–173.
- [27] *AES implementation in C*
<http://gladman.plushost.co.uk/oldsite/>, Brian Gladman.

- [28] C. E. Shannon, "Communication in the Presence of Noise," in *Proceedings of Institute of Radio Engineers*, vol. 37, 1949, pp. 10–21.
- [29] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid, "Task management in mpsoCs: an asip approach," ser. ICCAD'09.
- [30] K.-J. Chang and Y.-Y. Chen, "System-level fault injection in system design platform," in *ISIS 2007 PROCEEDINGS OF THE 8TH SYMPOSIUM ON ADVANCED INTELLIGENT SYSTEMS*, 2007, pp. 354–359.
- [31] Y. Cheng, M. Anguo, and M. Zhang, "Accurate and simplified prediction of l2 cache vulnerability for cost-efficient soft error protection," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 1, pp. 56–66, 2012.
- [32] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 603–608.
- [33] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 113.
- [34] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 555–560.
- [35] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Design Automation Conference (DAC)*, 2013, pp. 1–10.
- [36] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 381–386.
- [37] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.
- [38] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.
- [39] A. K. Coskun, T. S. Rosing, and K. C. Gross, "Temperature management in multiprocessor socs using online learning," ser. DAC '08.
- [40] D. R. Cox and D. V. Hinkley, *Theoretical statistics*. CRC Press, 1979.
- [41] D. Brooks, V. Tiwari, M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000, pp. 83–94.
- [42] D. Kammler, J. Guan, G. Ascheid, R. Leupers and H. Meyr, "A fast and flexible Platform for Fault Injection and Evaluation in Verilog-based Simulations," in *Proc. 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2009.

- [43] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
- [44] A. Das, A. Kumar, and B. Veeravalli, "Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems," in *Conference on Design, Automation and Test in Europe*, ser. DATE '13.
- [45] A. DeHon, H. M. Quinn, and N. P. Carter, "Vision for cross-layer optimization to address the dual challenges of energy and reliability," in *DATE*. IEEE, 2010, pp. 1017–1022.
- [46] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.
- [47] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Fifth ACM/IEEE International Symposium on Networks-on-Chip*, ser. NOCS '11.
- [48] L. Dirk, M. E. Nelson, J. F. Ziegler, A. Thompson, and T. H. Zabel, "Terrestrial thermal neutrons," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 6, pp. 2060–2064, 2003.
- [49] J. Donald and M. Martonosi, "Techniques for multicore thermal management: Classification and new exploration," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 78–88.
- [50] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcam technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 57.
- [51] E. Dubrova, "Fault tolerant design: An introduction," *Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden*, 2008.
- [52] E. Berlekamp, R. McEliece and H. van Tilborg, "On the Inherent Intractability of Certain Coding Problems," *IEEE Trans. on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.
- [53] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.
- [54] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert *et al.*, "Asim: A performance model framework," *Computer*, vol. 35, no. 2, pp. 68–76, 2002.
- [55] Y. Emre and C. Chakrabarti, "Techniques for compensating memory errors in JPEG2000," *IEEE Trans. VLSI Syst.*, vol. 21, no. 1, pp. 159–163, 2013. Online: <http://dx.doi.org/10.1109/TVLSI.2011.2180407>
- [56] L. Entrena, C. López, and E. Olías, "Automatic generation of fault tolerant vhdl designs in rtl," in *FDL (Forum on Design Languages)*. Citeseer, 2001.
- [57] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 7–18.

- [58] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 301–312.
- [59] *UMC Free Library 90 nm Process*
<http://freelibrary.faraday-tech.com/>, Faraday.
- [60] M. Faruque, V. Dinavahi, M. Steurer, A. Monti, K. Strunz, J. Martinez, G. Chang, J. Jatskevich, R. Iravani, and A. Davoudi, "Interfacing issues in multi-domain simulation tools," *Power Delivery, IEEE Transactions on*, vol. 27, no. 1, pp. 439–448, 2012.
- [61] E. Fernandez-Alonso, D. Castells-Rufas, J. Joven, and J. Carrabina, "Survey of noc and programming models proposals for mp soc," *International Journal of Computer Science Issues*, vol. 9, no. 2, pp. 22–32, 2012.
- [62] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. IEEE, 1998, pp. 284–293.
- [63] E. Fuchs, "An evaluation of the error detection mechanisms in mars using software-implemented fault injection," in *Dependable Computing—EDCC-2*. Springer, 1996, pp. 73–90.
- [64] G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer, "Investigation of increased multi-bit failure rate due to neutron induced seu in advanced embedded srams," in *2007 IEEE Symposium on VLSI Circuits*, 2007.
- [65] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and avf estimation revisited," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 477–486.
- [66] B. Gill, N. Seifert, and V. Zia, "Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node," in *Reliability Physics Symposium, 2009 IEEE International*. IEEE, 2009, pp. 199–205.
- [67] K. Gilles, "The semantics of a simple language for parallel programming," in *In Information Processing'74: Proceedings of the IFIP Congress*, vol. 74, 1974, pp. 471–475.
- [68] R. C. Gonzalez, *Digital image processing*. Pearson Education India, 2009.
- [69] M. S. Gordon, P. Goldhagen, K. P. Rodbell, T. Zabel, H. Tang, J. Clem, and P. Bailey, "Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, pp. 3427–3434, 2004.
- [70] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 409–414.
- [71] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE, pp. 3–14. Online: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=990739>

- [72] H. Cho, L. Leem and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 546–558, 2012.
- [73] M. W. H. Keding, F. Hürtgen and M. Coors, "Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Toronto, sep 1998.
- [74] A. Haghdooost, H. Asadi, and A. Baniasadi, "System-level vulnerability estimation for data caches," in *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on.* IEEE, 2010, pp. 157–164.
- [75] D. Handke, "Graphs with distance guarantees," Ph.D. dissertation, 1999.
- [76] F. Harary and J. P. Hayes, "Edge fault tolerance in graphs," *Networks*, vol. 23, no. 2, pp. 135–142, 1993.
- [77] F. Harary and J. P. Hayes, "Node fault tolerance in graphs," *Networks*, vol. 27, no. 1, pp. 19–23, 1996.
- [78] A. S. Hartman and D. E. Thomas, "Lifetime improvement through runtime wear-based task mapping," ser. CODES+ISSS '12.
- [79] R. B. Hitchcock Sr, "Timing verification and the timing analysis program," in *Proceedings of the 19th Design Automation Conference.* IEEE Press, 1982, pp. 594–604.
- [80] M. Hosseinabady, M. R. Kakoei, J. Mathew, and D. K. Pradhan, "Reliable network-on-chip based on generalized de bruijn graph," in *High Level Design Validation and Test Workshop, 2007. HLVDVT 2007. IEEE International.* IEEE, 2007, pp. 3–10.
- [81] *Documentation*
<http://lava.cs.virginia.edu/HotSpot/documentation.htm>, HotSpot 6.0.
- [82] M.-y. Hsieh, A. Rodrigues, R. Riesen, K. Thompson, and W. Song, "A framework for architecture-level power, area, and thermal simulation and its application to network-on-chip design exploration," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 63–68, 2011.
- [83] T. Huang, G. Yang, and G. Tang, "A fast two-dimensional median filtering algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 27, no. 1, pp. 13–18, 1979.
- [84] W. Huang, S. Ghosh, K. Sankaranarayanan, K. Skadron, and M. R. Stan, "Hotspot: Thermal modeling for cmos vlsi systems," *IEEE Transactions on Component Packaging and Manufacturing Technology*, pp. 200–205, 2005.
- [85] Q. Huynh-Thu and M. Ghanbari, "Scope of validity of psnr in image/video quality assessment," *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008.
- [86] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Scaling effects on neutron-induced soft error in srams down to 22nm process," in *Third Workshop on Dependable and Secure Nanocomputing*, 2009.
- [87] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule," *Electron Devices, IEEE Transactions on*, vol. 57, no. 7, pp. 1527–1538, 2010.

- [88] J. A. Clark and D. K. Pradhan, "Fault injection: a method for validating computer-system dependability," *IEEE Computer*, vol. 28, no. 6, pp. 47–56, 1995.
- [89] J. Srinivasan, S. V. Adve, P. Bose and J. A. Rivers, "The Case for Lifetime Reliability-Aware Microprocessors," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 276, 2004.
- [90] M. Jeitler, M. Delvai, and S. Reichör, "Fuse-a hardware accelerated hdl fault injection tool," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*. IEEE, 2009, pp. 89–94.
- [91] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: The mefisto tool," in *FTCS*, 1994, pp. 66–75.
- [92] J. S. JESD89A, "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices," *JEDEC solid state technology association*, 2006.
- [93] K. Amir and B. Eric, "Fast, Minimal Decoding Complexity, System Level, Binary Systematic (41, 32) Single-Error-Correcting Codes For On-chip DRAM Applications," in *Proceedings of the 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 308–313.
- [94] G. Kahn, "The semantics of simple language for parallel programming." in *IFIP Congress '74*.
- [95] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 820–825.
- [96] K. Kanda, K. Nose, H. Kawaguchi, and T. Sakurai, "Design impact of positive temperature dependence on drain current in sub-1-v cmos vlsi," *Solid-State Circuits, IEEE Journal of*, vol. 36, no. 10, pp. 1559–1564, 2001.
- [97] G. Karakonstantis, D. Mohapatra, and K. Roy, "Logic and Memory Design Based on Unequal Error Protection for Voltage-scalable, Robust and Adaptive DSP Systems," *Signal Processing Systems*, vol. 68, no. 3, pp. 415–431, 2012.
- [98] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE micro*, no. 1, pp. 8–11, 1994.
- [99] J. Kiusalaas, *Numerical methods in engineering with MATLAB®*. Cambridge university press, 2010.
- [100] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010.
- [101] K. Krewell, "Intel's mckinley comes into view," *Microprocessor Report*, vol. 15, no. 10, p. 1, 2001.
- [102] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an under-designed multiplier architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*. IEEE, 2011, pp. 346–351.
- [103] L. Myint and P. Supnithi, "Unequal Error Correction Strategy for Magnetic Recording Systems with Multi-track Processing," *J. Appl. Phys.*, vol. 111, 2012.

- [104] J. H. Lala and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 25–40, 1994.
- [105] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.
- [106] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha, "A task remapping technique for reliable multi-core embedded systems," ser. CODES/ISSS '10.
- [107] I. Lee, J. Kwon, J. Park, and J. Park, "Priority based error correction code (ECC) for the embedded SRAM memories in H.264 system," *Signal Processing Systems*, vol. 73, no. 2, pp. 123–136, 2013. Online: <http://dx.doi.org/10.1007/s11265-013-0736-4>
- [108] J. Lee and A. Shrivastava, "A Compiler-Microarchitecture Hybrid Approach to Soft Error Reduction for Register Files," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 7, pp. 1018–1027, 2010.
- [109] R. Leupers and O. Temam, *Processor and System-on-Chip Simulation*. Springer, Sep 2010, ISBN: 978-1-44196-174-7.
- [110] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.
- [111] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System implications of memory reliability in exascale computing," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 46.
- [112] S. Lin and D. Costello, Jr., *Error Control Coding: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, NJ., 1983.
- [113] A. Lingamneni, C. Enz, J.-L. Nagel, K. Palem, and C. Piguet, "Energy parsimonious circuit design through probabilistic pruning," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.
- [114] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Parsimonious circuits for error-tolerant applications through probabilistic logic minimization," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*. Springer, 2011, pp. 204–213.
- [115] B. Littlewood, "The impact of diversity upon common mode failures," *Reliability Engineering & System Safety*, vol. 51, no. 1, pp. 101–113, 1996.
- [116] W. Lu and M. Radetzki, "Efficient fault simulation of systemc designs," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 487–494.
- [117] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

- [118] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *In Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008.
- [119] M. Michael and D. Große and R. Drechsler, "Analyzing dependability measures at the Electronic System Level," in *Forum on Specification & Design Languages*, ser. FDL '11.
- [120] M. Sugihara, T. Ishihara, K. Hashimoto and M. Muroyama, "A Simulation-based Soft Error Estimation Methodology for Computer Systems," in *7th International Symposium on Quality Electronic Design*, 2006, march 2006.
- [121] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "Rifle: A general purpose pin-level fault injector," in *Dependable Computing—EDCC-1*. Springer, 1994, pp. 197–216.
- [122] W. Mansour and R. Velazco, "An automated seu fault-injection method and tool for hdl-based designs," *Nuclear Science, IEEE Transactions on*, vol. 60, no. 4, pp. 2728–2733, 2013.
- [123] W. Mansour and R. Velazco, "Seu fault-injection in vhdl-based processors: A case study," *Journal of Electronic Testing*, vol. 29, no. 1, pp. 87–94, 2013.
- [124] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," ser. MICRO 40.
- [125] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 2013, pp. 779–786.
- [126] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2012, pp. 728–735.
- [127] S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, "Rethinking error injection for effective resilience," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 390–393.
- [128] S. Mirkhani, S. Mitra, C.-Y. Cher, and J. Abraham, "Efficient soft error vulnerability estimation of complex designs," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 103–108.
- [129] S. Misera, H. T. Vierhaus, and A. Sieber, "Fault injection techniques and their accelerated simulation in systemc," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 587–595.
- [130] S. Mitra, N. R. Saxena, and E. J. McCluskey, "A design diversity metric and analysis of redundant systems," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 498–510, 2002.
- [131] S. Mitra, N. R. Saxena, and E. J. McCluskey, "Efficient design diversity estimation for combinational circuits," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1483–1492, 2004.
- [132] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 286–296.

- [133] G. E. Moore, "Cramming more components onto integrated circuits, volume 38, number 8, april 19, 1965, pp. 114 ff." *IEEE Solid-State Circuits Newsletter*, vol. 3, no. 20, pp. 33–35, 2006.
- [134] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings. 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99–110.
- [135] S. S. Mukherjee, C. T. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO*, 2003, pp. 29–42.
- [136] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.
- [137] M. Omana, G. Papasso, D. Rossi, and C. Metra, "A model for transient fault propagation in combinatorial logic," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 111–115.
- [138] K. Palem and A. Lingamneni, "Ten years of building broken chips: The physics and engineering of inexact computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 87:1–87:23, May 2013. Online: <http://doi.acm.org/10.1145/2465787.2465789>
- [139] Y. Park, S. Pasricha, F. Kurdahi, and N. Dutt, "A multi-granularity power modeling methodology for embedded processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 4, pp. 668–681, 2011.
- [140] S. Paul, F. Cai, X. Zhang, and S. Bhunia, "Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache," *IEEE Trans. Computers*, vol. 60, no. 1, pp. 20–34, 2011.
- [141] W. W. Peterson and E. J. Weldon, *Error-correcting codes*. MIT press, 1972.
- [142] Processor Designer: C Compiler Reference Manual 2013.06.
- [143] R. Hegde and N.R. Shanbhag, "Energy-efficient Signal Processing via Algorithmic Noise-tolerance," ser. ISLPED '99.
- [144] R. Naseer, "Parallel Double Error Correcting Code Design to Mitigate Multi-bit Upsets in SRAMs," in *ESSCIRC*, 2008, pp. 222–225.
- [145] R. Velazco, S. Karoui, T. Chapuis, D. Benezech and L.H. Rosier, "Heavy Ion Tests for the 68020 Microprocessor and the 68882 Coprocessor," in *IEEE Transactions on Nuclear Science*, vol. 39, no. 3, 1992.
- [146] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal*, vol. XXVI, no. 2, pp. 147–160, 1950.
- [147] B. Rahbaran, A. Steininger, and T. Handl, "Built-in fault injection in hardware-the fidyco example," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 327–332.
- [148] A. Rajendiran, S. Ananthanarayanan, H. D. Patel, M. V. Tripunitara, and S. Garg, "Reliable computing with ultra-reduced instruction set co-processors," in *Proceedings of the 49th Design Automation Conference*, 2012, pp. 697–702.

- [149] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: synthesis of approximate sequential circuits," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 364.
- [150] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: embedded code generation aiming at reliability," in *CODES+ISSS*, 2011, pp. 237–246.
- [151] S. Rehman, A. Toma, F. Kriebel, M. Shafique, J.-J. Chen, and J. Henkel, "Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 273–282.
- [152] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.
- [153] R. Riter, "Modeling and testing a critical fault-tolerant multi-process system," in *Fault-Tolerant Computing, FTCS-25.*, 1995, pp. 516–521.
- [154] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *FTCS*. IEEE COMPUTER SOCIETY PRESS, 1999, pp. 84–93.
- [155] S. Ahmed, "Unequal Error Protection Using Fountain Codes With Applications to Video Communication," *IEEE Trans. on Multimedia*, vol. 13, no. 1, pp. 92–101, 2011.
- [156] S. Baloch, T. Arslan and A. Stoica, "Efficient Error Correcting Codes for On-Chip DRAM Applications for Space Missions," in *IEEE Aerospace*, 2005, pp. 1–9.
- [157] S. Borade, B. Nakiboglu and L. Zheng, "Unequal Error Protection: An Information-theoretic Perspective," *IEEE Transactions on Information Theory*, vol. 55, no. 12, pp. 5511–5539, 2009.
- [158] S. K. Reinhardt and S. S Mukherjee, "Transient Fault Detection via Simultaneous Multi-threading," ser. ISCA '00, 2000, pp. 25–36.
- [159] S. Krishnaswamy, G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Accurate Reliability Evaluation and Enhancement via Probabilistic Transfer Matrices," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 282–287.
- [160] S. Krishnaswamy, G. F. Viamontes, I. L. Markov and J. P. Hayes, "Probabilistic transfer matrices in symbolic reliability analysis of logic circuits," *ACM Trans. Design Autom. Electr. Syst.*, vol. 13, no. 1, 2008.
- [161] S. Rehman, M. Shafique and J. Henkel, "Instruction scheduling for reliability-aware compilation," ser. DAC '12. ACM, 2012, pp. 1292–1300. Online: <http://doi.acm.org/10.1145/2228360.2228601>
- [162] S. Rehman, M. Shafique, F. Kriebel and J. Henkel, "RAISE: Reliability-Aware Instruction Scheduling for unreliable hardware," in *ASP-DAC '12*, 30 2012-feb. 2 2012, pp. 671–676.
- [163] T. Sakurai *et al.*, "Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas," *Solid-State Circuits, IEEE Journal of*, vol. 25, no. 2, pp. 584–594, 1990.

- [164] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.
- [165] A. Sassone, A. Calimera, A. Macii, E. Macii, M. Poncino, R. Goldman, V. Melikyan, E. Babayan, and S. Rinaudo, "Investigating the effects of inverted temperature dependence (ITD) on clock distribution networks," in *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012*, 2012, pp. 165–166.
- [166] L. Schor, I. Bacivarov, D. Rai, H. Yang, S. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," ser. CASES '12.
- [167] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, "Soft error susceptibilities of 22 nm tri-gate devices," *Nuclear Science, IEEE Transactions on*, vol. 59, no. 6, pp. 2666–2673, 2012.
- [168] T. Semiconductor, "Soft errors in electronic memory-a white paper," 2004.
- [169] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "Systemc-based minimum intrusive fault injection technique with improved fault representation," in *On-Line Testing Symposium, 2008. IOLTS'08. 14th IEEE International*. IEEE, 2008, pp. 99–104.
- [170] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 17.
- [171] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 389–398.
- [172] V. Sieh, O. Tschache, and F. Balbach, "Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. IEEE, 1997, pp. 32–36.
- [173] A. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," ser. DAC '13.
- [174] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 2–13, 2003.
- [175] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *CHES*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523. Springer, 2002, pp. 2–12.
- [176] C. Slayman, "Soft errors—past history and recent discoveries," in *Integrated Reliability Workshop Final Report (IRW), 2010 IEEE International*. IEEE, 2010, pp. 25–30.
- [177] J. Somers, F. Director, and S. Graham, "Stratus ftserver–intel fault tolerant platform," in *Rap. tech. Intel Developer Forum, Fall, 2002*.

- [178] V. Soteriou, N. Eisley, H. Wang, B. Li, and L.-S. Peh, "Polaris: a system-level roadmapping toolchain for on-chip interconnection networks," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, no. 8, pp. 855–868, 2007.
- [179] L. Spainhower and T. A. Gregg, "Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5.6, pp. 863–873, 1999.
- [180] D. A. Spielman, "Linear-Time Encodable and Decodable Error-Correcting Codes," *IEEE Transactions on Information Theory*, vol. 42, pp. 388–397, 1996.
- [181] *Design Compiler*
http://www.synopsys.com/products/logic/design_compiler.html, Synopsys.
- [182] *Processor Designer*
<http://www.synopsys.com/Systems/BlockDesign/processorDev>, Synopsys.
- [183] X. Tang, V. K. De, and J. D. Meindl, "Intrinsic mosfet parameter fluctuations due to random dopant placement," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 5, no. 4, pp. 369–376, 1997.
- [184] <http://www.target.com>, Target Compiler Technologies.
- [185] F. Terraneo, D. Zoni, and W. Fornaciari, "An accurate simulation framework for thermal explorations and optimizations," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2015, p. 5.
- [186] *CoSy Compiler Development System*
www.ace.nl/compiler/cosy.html, The ACE Companies.
- [187] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, 1994.
- [188] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, "Instruction level power analysis and optimization of software," *The Journal of VLSI Signal Processing*, vol. 13, no. 2, pp. 223–238, 1996.
- [189] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 175–186, 2010.
- [190] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE, 2011, pp. 91–99.
- [191] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain *et al.*, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 29–41, 2008.
- [192] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 1–12.

- [193] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: a unified design paradigm for approximate and quality configurable circuits," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 1367–1372.
- [194] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 796–801.
- [195] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2011, pp. 667–673.
- [196] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 43–98.
- [197] S. Wang, "Characterizing system-level vulnerability for instruction caches against soft errors," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 356–363.
- [198] Z. Wang, C. Chen, P. Sharma, and A. Chattopadhyay, "System-level reliability exploration framework for heterogeneous mpsoc," in *Proceedings of the 24th edition of the great lakes symposium on VLSI*. ACM, 2014, pp. 9–14.
- [199] Z. Wang, R. Li, and A. Chattopadhyay, "Opportunistic redundancy for improving reliability of embedded processors," in *8th International Design and Test Symposium, IDT 2013, Marrakesh, Morocco, 2013*, pp. 1–6.
- [200] Z. Wang, G. Paul, and A. Chattopadhyay, "Processor design with asymmetric reliability," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*. IEEE, 2014, pp. 565–570.
- [201] Z. Wang, H. Xie, S. Chafekar, R. U. A. Sai, and A. Chattopadhyay, "Architectural error prediction using probabilistic error masking matrices," in *Asia Symposium on Quality Electronic Design (ASQED)*. IEEE, 2015.
- [202] Z. Wang, L. Yang, and A. Chattopadhyay, "Architectural reliability estimation using design diversity," in *Quality Electronic Design (ISQED), 2015 16th International Symposium on*. IEEE, 2015, pp. 112–117.
- [203] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 411–420.
- [204] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 264.
- [205] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-I. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 83–93, 2010.

- [206] E. M. Witte, A. Chattopadhyay, O. Schliebusch, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr, "Applying resource sharing algorithms to adl-driven automatic asip implementation," in *ICCD 2005*, 2005, pp. 193–199.
- [207] A. Wood, R. Jardine, and W. Bartlett, "Data integrity in hp nonstop servers," in *Workshop on SELSE*, 2006.
- [208] W. Yueh, M. Cho, and S. Mukhopadhyay, "Perceptual quality preserving SRAM architecture for color motion pictures," in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 2013, pp. 103–108. Online: <http://dl.acm.org/citation.cfm?id=2485315>
- [209] Z. Wang, C. Chen and A. Chattopadhyay, "Fast reliability exploration for embedded processors via high-level fault injection," in *ISQED*, 2013, pp. 265–272.
- [210] Z. Wang, K. Singh, C. Chen and A. Chattopadhyay, "Accurate and efficient reliability estimation techniques during ADL-driven embedded processor design," in *DATE*, 2013, pp. 547–552.
- [211] Z. Wang, R. Li and A. Chattopadhyay, "Opportunistic redundancy for improving reliability of embedded processors," in *8th IEEE International Design and Test Symposium (IDT)*, 2013.
- [212] H. R. Zarandi, S. G. Miremadi, and A. Ejlali, "Fault injection into verilog models for dependability evaluation of digital systems," in *null*. IEEE, 2003, p. 281.
- [213] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *Electron Devices, IEEE Transactions on*, vol. 53, no. 11, pp. 2816–2823, 2006.
- [214] H. Zheng, L. Fan, and S. Yue, "Fitvs: A fpga-based emulation tool for high-efficiency hardness evaluation," in *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*. IEEE, 2008, pp. 525–531.

Curriculum Vitae

Name	Zheng Wang
Date of birth	13 Dec. 1983
Place of birth	Tianjin, China

since Oct. 2015	Research associate at School of Electrical and Electronic Engineering Nanyang Technological University, Singapore
-----------------	--

Sept. 2010 – Sept. 2015	Research assistant at Institute of Multi-Processor System-on-Chip Architectures (MPSoC) RWTH-Aachen University, Germany
-------------------------	--

Oct. 2007 – Sept. 2009	Master of Science at Institute for Electronic Design Automation (EDA) Technische Universität München, Germany Master thesis at Infineon Technology, Munich, Germany “Pfair Scheduling Algorithm on ARM Multiprocessor”
------------------------	---

Sept. 2002 – Aug. 2007	Bachelor of Science at Department of Physics Shanghai Jiao Tong University, China
------------------------	--