Computer Architecture and Design Methodologies

Zheng Wang
Anupam Chattopadhyay

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

Springer

Computer Architecture and Design Methodologies

Series editors

Anupam Chattopadhyay, Noida, India
Soumitra Kumar Nandy, Bangalore, India
Jürgen Teich, Erlangen, Germany
Debdeep Mukhopadhyay, Kharagpur, India

Twilight zone of Moore's law is affecting computer architecture design like never before. The strongest impact on computer architecture is perhaps the move from unicore to multicore architectures, represented by commodity architectures like general purpose graphics processing units (gpgpus). Besides that, deep impact of application-specific constraints from emerging embedded applications is presenting designers with new, energy-efficient architectures like heterogeneous multi-core, accelerator-rich System-on-Chip (SoC). These effects together with the security, reliability, thermal and manufacturability challenges of nanoscale technologies are forcing computing platforms to move towards innovative solutions. Finally, the emergence of technologies beyond conventional charge-based computing has led to a series of radical new architectures and design methodologies.

The aim of this book series is to capture these diverse, emerging architectural innovations as well as the corresponding design methodologies. The scope will cover the following.

Heterogeneous multi-core SoC and their design methodology

Domain-specific Architectures and their design methodology

Novel Technology constraints, such as security, fault-tolerance and their impact on architecture design

Novel technologies, such as resistive memory, and their impact on architecture design

Extremely parallel architectures

More information about this series at http://www.springer.com/series/15213

Zheng Wang · Anupam Chattopadhyay

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

Springer

Zheng Wang
Shenzhen Institutes of Advanced Technology
Chinese Academy of Sciences
Shenzhen
China

Anupam Chattopadhyay
School of Computer Science and Engineering
Nanyang Technological University
Singapore
Singapore

ISSN 2367-3478 ISSN 2367-3486 (electronic)
Computer Architecture and Design Methodologies
ISBN 978-981-10-1072-9 ISBN 978-981-10-1073-6 (eBook)
DOI 10.1007/978-981-10-1073-6

Library of Congress Control Number: 2017943095

© Springer Science+Business Media Singapore 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Acknowledgements

This book is the result of my work as research associate 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. Dr. -Ing. Anupam Chattopadhyay and Prof. Dr. -Ing. Tobias Noll. Prof. Chattopadhyay has been extremely helpful and tremendously inspiring throughout my Ph.D. 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 at 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 book and my life in general. And finally, infinite gratitude to my beloved wife as well as my son.

December 2016 Zheng Wang

Contents

List of Figures

List of Tables

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 devices, circuits, 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 book, 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 statistical fault injection scheme is designed considering dynamic timing analysis of the architecture. A power/thermal/timing error co-simulation framework is demonstrated for integrating fault injection with the 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 design ad-hoc fault tolerant network for arbitrary task graph with the optimal amount of connecting edges is elaborated and verified by exhaustive search based algorithm.

The methodologies proposed in this book 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.

Chapter 1
Introduction

The last few decades have witnessed continuous scaling of CMOS technology, guided by Moore's Law [136], 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) [10], 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 [55]. Second, increasing power dissipation leads to thermal stress which affects design lifetime as well as soft error rates [39]. Third, continuous technology scaling gives rise to increased permanent errors caused by process variation [185]. Fourth, frequency and voltage over-scaling targeting timing margin exploration save power and performance budgets but also introduce timing errors [59]. Finally, new kinds of innovative fault-based attacks against cryptographic modules [176] 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 modeled at low levels of design abstractions. For instance, at device level the vulnerability of a transistor against striking particles is analyzed according to its physical properties. The deviation of 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.

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_1

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 [59]. An uncorrected data error representing the color of a pixel in an image can be intrinsically tolerated by the limit of human perception [15].

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 book, multiple challenges for developing the reliability estimation and exploration framework is 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 Chap. 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

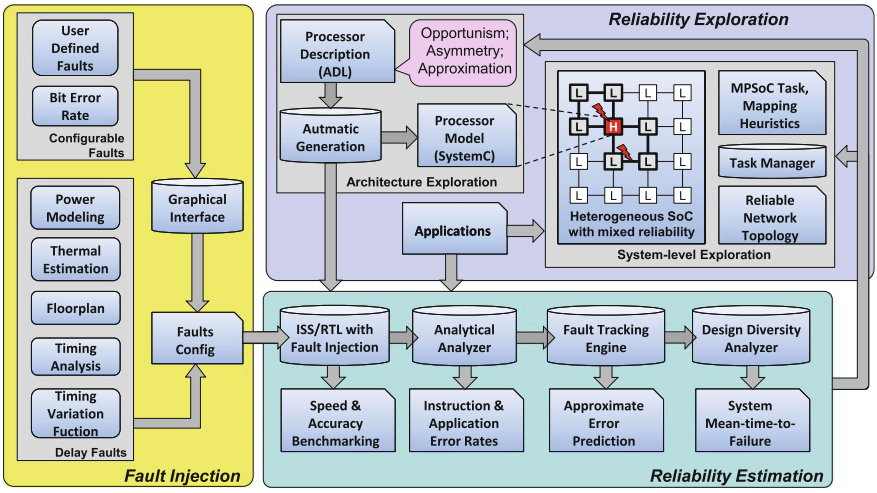


Fig. 1.1 Overall flow of high-level reliability estimation and exploration

cells while achieving 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 a graphical interface. In the timing mode, logic delay faults are injected based on the statistics from low-level timing analysis and variation function. In Sect. 4.2 the fault injector is extended for system-level modules described in SystemC language. Section 4.3 illustrates the approach to model timing error from instruction dependent statistical data under voltage scaling and noise, which is applied to the analysis of application performance. Another interesting case study is to relate delay error injection with power consumption and runtime temperature variation, therefore a joint simulation framework for power, temperature and delay faults is proposed in Sect. 4.4.

- **High-level Reliability Estimation**
  Architectural reliability can be fast estimated through analytical methods, which are discussed in Chap. 5. Section 5.1 presents an analytical estimation technique based on a 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 Sect. 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 Sect. 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 Chap. 6. The first technique, named as opportunistic redundancy in Sect. 6.1, introduces a passive error detection policy for algorithmic units by re-executing the instruction only if there exist underutilized resources, which incurs a very small performance penalty. The approach is benchmarked with an aggressive policy where all instructions are double executed to verify the correctness of results. The second technique, named as asymmetric redundancy in Sect. 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 Sect. 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 Chap. 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 design an ad-hoc fault tolerant network for arbitrary task graph, which contains an optimal amount of connecting edges. An 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 book 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 book are presented in Chap. 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

Reliabilty is in a broad sense one attribute of *dependability*, which describes the ability of the system to deliver its intended service [53]. 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

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_2

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 [210] 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 book concentrates 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 [140].
- ***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 [7].

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 [55].
- ***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 [50].

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 1–10% 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 45 nm technology [19].

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 book 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) [94]. Recently, MBU and MCU become important threats for nanoscale technologies [88]. 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:

$$MTTF_{sys} = \frac{1}{\sum_{i=1}^{n} \frac{1}{MTTF_i}} \tag{2.1}$$

- ***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:

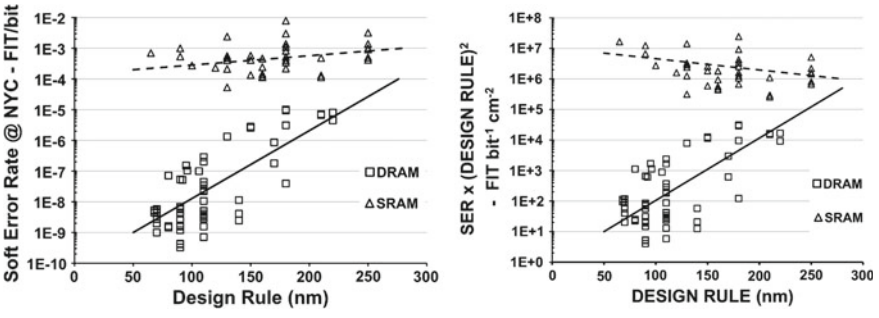
$$FIT_{sys} = \sum_{i=1}^{n} FIT_i \tag{2.2}$$

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 (Fig. 2.1a and b) and combinatorial logic (Fig. 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



(a) SER vs Design rule (b) Per unit area SER vs Design rule

Fig. 2.1 SER scale trend for SRAM and DRAM [177] Copyright ©2010 **IEEE**

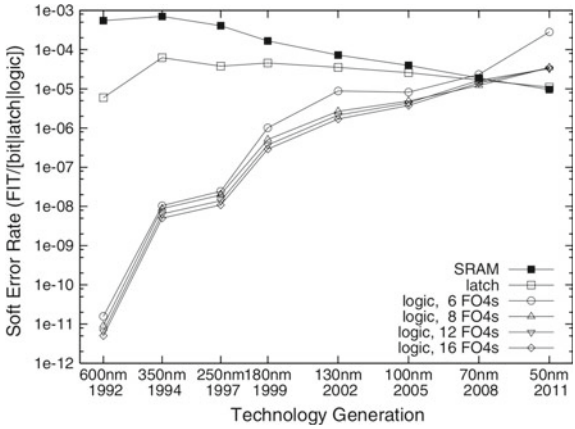


Fig. 2.2 SER scale trend for combinatorial logic [172] Copyright ©2002 **IEEE**

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 250 nm to 50% at 22 nm [89]. The work in [66] also investigates the MBU rate for 65 nm.

- ***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.
- ***Combinatorial logic*** Fig. 2.2 shows the predicted trend of logic SER rate from Shivakumar [172] from 600 nm till 50 nm 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 [68] presents that the logic SER is below 30% of nominal latch SER for 32 nm fabricated chips.

Chapter 3
State-of-the-Art

In this chapter, the related work of this book is elaborated. Initially, fault injection techniques are discussed. Following that, major high-level reliability estimation techniques are briefly illustrated. Afterward, 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 a tolerated range of faults, which is documented in the device specification.
- Explore efficient fault tolerant techniques in a specific faulty environment.
- Estimate fault coverage of testing mechanism in the device.
- Understand the behavior of real physical faults and benchmark with high-level fault injection techniques.

Hardware related FI techniques are the focus of this book. 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.

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_3

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*** [8] 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*** [124] 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 analyze the protection efficiency of multiple fault tolerant modules.
- ***FIST*** [100] create transient faults to the system using both contact and non-contact techniques. The device is exposed to 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 an MOS transistor locating between the power line and the V_{cc} pin to mimic the power fluctuation.
- ***MARS*** [65] 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 the device.
- ***Van@2011*** [192] 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 microcontroller.

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 a chip. Recently, the emergence of virtual prototyping also shows its usage in system-level design for reliability purpose [201]. 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 a significant role in fault simulation and analysis.

In general, techniques for simulated fault injection are classified into *simulator commands* (SC) and *code-modification* (CM). Most common techniques in CM are *Saboteurs* and *Mutant* [16, 93]. Saboteurs add extra components to the original RTL modes, while Mutant substitutes the original model with the modified one. Both techniques are limited to the fact that model recompilation is always required which is time-consuming. On the other hand, SC-based techniques apply the simulator commands to dynamically update the resource values in the design without model recompilation. The major issue in SC techniques is the controllability of the simulator commands since usually not all of the resources within the hardware architectures can be controlled. Besides, the portability among various simulators raises one extra issue.

In this section, representative simulated FI approaches are listed 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*** [173] 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*** [24] uses Saboteurs technique to augment the original VHDL module with fault injection capabilities. Automated paring, fault injection, and result extraction blocks are designed to speed up fault simulation. Another tool variation is named MEFISTO-C [64], which applies simulator command method to inject fault on the fly. The Vantage Optimum VHDL Simulator has been used for parallel simulation on the network of UNIX workstations.

- ***GSTF*** [16] 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 analyze the result from fault configurations in order to validate the fault tolerant mechanisms.
- ***FIT*** [58] 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*** [18] 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*** [218] 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*** [44] 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 work on the simulator models described in high-level languages such as C++ and SystemC. It provides an efficient solution for the design of fault tolerant techniques in MPSoC architectures.

- ***Chang@2007*** [29] presents the Saboteurs based FI technique for SystemC and demonstrates the usability for different levels of Transaction Level Models (TLM).
- ***Misera@2007*** [132] 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*** [170] 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 the simulation with new data types.
- ***Beltrame@2009*** [17] 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 the command line or parsed from a fault configuration XML file.

- ***Lu@2011*** [119] 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 of 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*** [149] 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 the 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*** [3] 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*** [220] 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 the flip-flop into 8 gates implementation for FI.
- ***FuSE*** [92] 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 a huge number of FI experiments are realized simultaneously.
- ***FLIPPER*** [6] presents the FPGA emulation platform for SEU in the configuration memory. Proton irradiation is performed for FI during the ground test. The effects of various protection mechanism are tested in the radiation environment.
- ***DFI*** [126] 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 on the host PC from Ethernet link port. FI can be performed in single clock cycle when processor runs an application.

- ***NETFI*** [125] 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 the 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 [130].

3.2 Analytical Reliability Estimation

Despite 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 book for further proposals.

3.2.1 Architecture Vulnerability Factor Analysis

Architecture Vulnerability Factor (AVF) was proposed in [138] 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 the 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 the microarchitectural perspective as idle state bits, mis-speculated state bits, predictor structure bits and Ex-ACE state bits. From the 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 [138] for details of un-ACE bits.

The calculation of ACE bits involves a performance simulator, where performance counters are used to profile and track the instructions. This is demonstrated using Asim framework [56] of IA64 architecture [103] 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* [108], 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 [199], data cache [76], L2 cache [30] and register file [135]. However, the author in [67] pointed out that a 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 bit flip model is advised to be replaced by flipping with a 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. The 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 nature of program. It is advised in Sect. 5.2 [207] of this book that ACE bits can be accurately identified through probabilistic fault tracking analysis in an 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. [153] proposed compiler optimization techniques to generate reliable code which minimizes the ACE latencies of the program variable. The work is further extended in [154] to jointly consider functional correctness and timing reliability. In [210] Both 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 Sect. 5.1 [216].

3.2.2 *Probablistic Transfer Matrix*

Krishnaswamy [162] 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 indicate 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]. The 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 serially connected circuits, the 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 input of the circuit with error probabilities. An extension of PTM algebra is also presented in [162] which models the electrical masking effects due to error glitch attenuation through the logic gates [140]. 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 [161] 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 Sect. 5.2 [207] 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 [12], is used to protect the redundant system from CMF by an independent generation of two or more hardware/software components. For instance, N-version programming [13] is applied to attain diversity. Hardware diversity is applied in the Primary Flight Computer (PFC) system of Boeing 777 [156] 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 a quantitative evaluation metric for the redundant system [133], 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 [133] 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 a reliability evaluation metric. Efficient diversity estimation techniques for combinatorial circuits are proposed in [134], which works on circuit structures showing regularity features. For the 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 [120]. 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 the 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 the 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 Sect. 5.3.1 [208].

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 [52]. 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 [160] 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 is 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 causes an 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*** [178] 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*** [213] 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*** [180] 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 [180] estimate an area overhead of 35% from this Lock-stepped implementation.

- ***AR-SMT*** [157] 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*** [11] 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 [209]. 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 the case of a mismatch, the result of the checker core is adopted. Transient faults in the checker core itself are not addressed. Besides, DIVA cannot detect the error from the decode stage.
- ***Argus*** [127] 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*** [150] realizes the RMT protection by a ultra-reduced instruction-set coprocessor, which has only one Turing complete instruction *subleq* from the MIPS ISA. Different instructions in the main core are 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 a 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 of the existing resources. The details are presented in Sect. 6.1 [217].

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 [169]. 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. The 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 the case of a

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 kindly referred to the book by Peterson and Weldon [144].

ECC has been investigated heavily for the mainstream processors. IBM introduces the concept of Chipkill-correct [48], 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 [20]. In [191] 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 [211]. An efficient implementation of BCH is presented in [114].

In Sect. 6.2 [205] 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 [74]. Furthermore, applications such as video and audio processing exhibit the feature of cognitive resilience due to the limitation of human perception. In [32] a 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 is illustrated.

3.3.2.1 Circuits-Level Techniques

- ***Kahng@2012*** [97] 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*** [72] 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 a 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*** [129] 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 the synthesis of lower significant bits. The proposed adder achieves up to 60% energy saving compared to the conventional timing-starved adder.
- ***Kulkarni@2011*** [104] 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*** [59] 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*** [38] 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 over-scaling without timing errors. Results show that 38% of speed increment or 24% power reduction is achieved.

3.3.2.2 Architectural Techniques

- ***ERSA*** [74] 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 the non-error-resilient portion of applications, while cheap RRC for portions of the 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*** [33] 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 the circuit level, dynamic precision control at the architectural level and significance-driven algorithmic truncation at the application level. Greater energy saving is gained by synergistically co-optimizing across different abstractions.
- ***Chippa@2011*** [31] 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 to the ones in [33].
- ***Georgios@2012*** [99] 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*** [15] designs a 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*** [196] 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*** [152] 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 the SALSA-based technique for synthesizing the combinational blocks.
- ***MACACO*** [197] proposes a systematic methodology to analyze 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 the behavior of the approximate circuit. SAT solver predicts the worst-case error while BDD gives the error distribution.

- ***GALS*** [128] 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*** [195] 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 are consequently downsized due to extra timing slack. The approach ensures the input quality constraints and iteratively performs substitution automatically.
- ***Probabilistic Pruning*** [116] 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 the conventional design.
- ***Probabilistic Logic Minimization*** [117] is another ranking based optimization technique during logic synthesis by intentionally bit-fliping 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 the conventional design.

3.3.2.4 Programming and Compilation Techniques

- ***EnerJ*** [166] 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*** [60] 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*** [194] 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*** [14] 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*** [171] 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 Sect. 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 the 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 multiprocessor 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 communi-

cate with each other. The problem of task mapping involves an optimal task deposition and scheduling mechanism in favor 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 the 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 [174] is referred to for details on the various mapping strategies.

Within a 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*, [46] discusses the proper approaches to address the lifetime optimization in terms of mean time to failure. Coskun et al. [41] discusses temperature-aware mapping that leads to increased lifetime. A wear-based heuristic is proposed in [80] 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 [109] the author proposes reliable remapping technique aiming at determining task migrations with the minimum cost while minimizing the throughput degradation. In [168] 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. [49] 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 [74] 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 Sect. 7.1 [201] 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 literature 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*** [4] 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 neighboring nodes, where d is a number of its neighboring nodes. Besides, the symmetric property of group graphs automatically alleviates many interconnection problems such as congestion and message-routing.
- ***Generalized de Bruijn Graph*** [82] 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*** [77] 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 numbers of edges fail.
- ***Node fault tolerance*** [79] 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 the 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*** [78] addresses a similar problem as in [79] 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 the tolerance of k-edge failures.

In Sect. 7.2, the methodology in [79] 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, high-level 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 Sect. 4.1. After that, the system-level fault injector is presented in Sect. 4.2. Two case studies of architecture fault injection techniques are presented. An application-level investigation on the impact of statistical timing errors under frequency/voltage scaling and voltage noise is illustrated in Sect. 4.3. Another error injection framework considering dynamic power and thermal impact is described in Sect. 4.4.

4.1 Architectural Fault Injection

Across various levels of design abstractions, faults can be accurately simulated only at circuits level due to the accurate physical modeling of faults, which is in turn extremely time-consuming. Many proposals have been developed to inject faults at the higher level of design abstractions as discussed in Sect. 3.1. Among those techniques, architectural fault injection plays an important role for design prototyping, where the states of an architecture, usually in register transfer level (RTL), are altered dynamically to imitate the behavior of physical faults [16, 44, 93]. However, RTL fault injection is relatively slow for the prototyping of modern complex SoC system. Furthermore, the modification of architecture states requires either the mutant of RTL design which needs repetitive model recompilation, or the extension of RTL simulator which is usually unavailable for major EDA tools.

Alternatives have appeared recently to simulate faults in high-level processor simulators, which offers a similar accuracy of RTL fault simulation. This is leveraged by the design of cycle-accurate instruction-set simulator which fully mimic the RTL behaviours [121, 123]. With the advantage in exploration speed, the major deficit

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_4

of such approach is the synergy with hardware descriptions. Any exploration on the instruction-set simulators requires sophisticated update on the targeted RTL codes, which provides extra difficulties in managing the team of both software and hardware designers.

The lack of generic tools for fault injection at the cycle-accurate architectural simulator and automatic exploration of hardware descriptions enforces the designer to include specific RTL codes or design environment in the design of fault tolerant processors, which impedes early design space exploration. This leads to the consequence that developed error detection and correction mechanisms are difficult to fully exploit the knowledge of architecture and software compilation techniques. This issue can be addressed only by including reliability at an early phase of design space exploration in the state-of-the-art architecture design and prototyping methodology.

A Large body of research work exists regarding high-level processor design. With the advent of Architecture Description Languages (ADLs), early design space exploration of digital processor becomes faster and easier with the ability of cycle-accurate simulation and automatic synthesizable RTL generation. This design methodology has proved to improve design quality and productivity tremendously, which has gained widespread commercial success [184, 186].

Contribution In this work, fault injection technique has been integrated into a commercial high-level processor design environment, where arbitrary processor architecture can be modeled and implemented. Such environment takes advantage of ADL LISA (Language for Instruction Set Architecture) [1]. With proposed high-level fault injector, the user can easily customize hardware or software based error protection mechanism and quickly evaluate its physical impact such as area, critical timing and power consumption. The approach achieves similar accuracy with Verilog-based fault injection technique. Although proposed by ADL LISA, the proposed technique is general to be applied to any high-level architecture design environment.

4.1.1 Methodologies

This section presents the fault injection methodologies, which includes the fault models, fault injection and error evaluation. A brief overview of LISA processor modeling language is first given to provide background information.

4.1.1.1 Brief Overview of LISA

LISA language describes the architecture and instruction encoding of application specific instruction-set processors (ASIP). The *OPERATION* section includes several primitives to represent the instruction set (*SYNTAX* and *CODING* primitives), operation scheduling (*ACTIVATION* primitive) and behaviour (*BEHAVIOR* primitive) of the processor. The *BEHAVIOR* section encloses plain C codes to specify

arbitrary functionalities. Legacy codes in RTL and SystemC formats are able to integrate as LISA operations through user APIs.

Architectural resources such as registers, memories, pipeline registers, memories, and pins are declared in the *RESOURCE* section. Such resources can be globally accessed from LISA operations. The language allows assigning operations into specific pipeline stages for scheduling through operations activation. Hence, the Instruction Set Architecture (ISA) is completely described by operations and resources.

From the LISA description, both software tools including a cycle-accurate instruction set simulator, C compiler, assembler/linker and synthesizable RTL descriptions are generated. Several optimization algorithms have been proposed to further optimize the physical footprint of generated hardware descriptions. Recently, the language extension to support LLVM compiler framework has been done. Interested readers are referred to [1, 184] for further information on LISA language.

4.1.1.2 Fault Models

For high-level fault simulation, physical faults in circuits are realized as the modification of logic states. Several basic types of logic faults are presented in Table 4.1. The fault time t happens between fault starting time t_s and fault ending time t_e. The resource value before the fault is shown as $V(t)$ and the one after the fault is as $F(t)$. The transient faults are set by a finite t_e while the permanent faults are configured by an infinite t_e. For instance, three kinds of bit-flip faults are implemented. Instantaneous bit-flip fault negates the resource value at the time of fault injection and gives the control of resource value to the behavioral simulator. Instantaneous bit-flip is used to model radiation induced faults such as SEU. Bit-flip with duration maintains the faulty value for a time window before releases its control to the simulator. The toggling bit-flip fault toggles the original resource value in specified time windows. Bit-flip with duration and toggling bit-flip can be used to characterize coupling faults, where resource value is dependent on another driving resources in a timing window. Each fault is labeled by an error rate which indicates the probability of fault happening. The fault models, which are realized as inherited data structure in object-oriented languages such as C++, are easily extensible by the users.

Table 4.1 Currently implemented fault types [215] Copyright ©2013 **IEEE**

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

The simple fault models do not cover complicated physical fault properties. However, the key characteristics such as the fault type, injection time, lasting duration and probability are used to link logic representation of faults with physical behaviors. Physical-aware fault modeling, which is an extension of the work in this section, are demonstrated by two use cases in Sect. 4.4 for aging faults and Sect. 4.3 for faults due to frequency/voltage over-scaling.

4.1.1.3 Fault Injection Method

As illustrated in Sect. 3.1.2, Simulator Command (SC) and Code Modification (CM) are two prevalent methods for simulation-based fault injection. The simulator generated by LISA language has programming interface support to facilitate SC method. However, similar to other SC based method, the LISA based SC method suffers from the scope of fault injection, where only globally available hardware resources are prone to state modification. The logic components inside operations such as local variables are uncontrollable by simulator commands. To overcome such inefficiencies, a hybrid approach combining both SC and CM methods has been adopted. Additional global signals, which can be automatically added to scripting languages, are used to control local variables through simulator commands. Consequently, the controllability of SC method is strongly enhanced. The user can switch between modes of fault injection into global resource only or into the LISA operations as well through model recompilation.

4.1.1.4 Error Evaluation Method

The *Error Manifestation Rate* (*EMR*) [44] is adopted to evaluate the impacts of faults. Assuming in each fault injection experiment one fault is randomly injected into the targeted architecture component, *EMR* quantifies the percentage of experiments which leads to errors on the component interfaces. Such error is defined as a mismatch of logic values in both time and locations with the fault-free experiment. *EMR* measures the ability of faults manifested into errors on component interfaces. Formally, given N_i as the total count of experiments,

$$EMR = N_e/N_i \times 100\% \tag{4.1}$$

where N_e is the count of experiments with errors. *EMR* tends to increase with the count and time window of faults. A larger *EMR* value implies higher error probability and thus less reliability of the targeted component. By defaults, EMR detects the error on interface signals. However, the user can define customized errors by comparing values of other interested signals. Note that *EMR* does not directly determine the percentage of user visible error which is highly dependent on the application characteristics. However, it reflects purely architecture level error resilience ability.

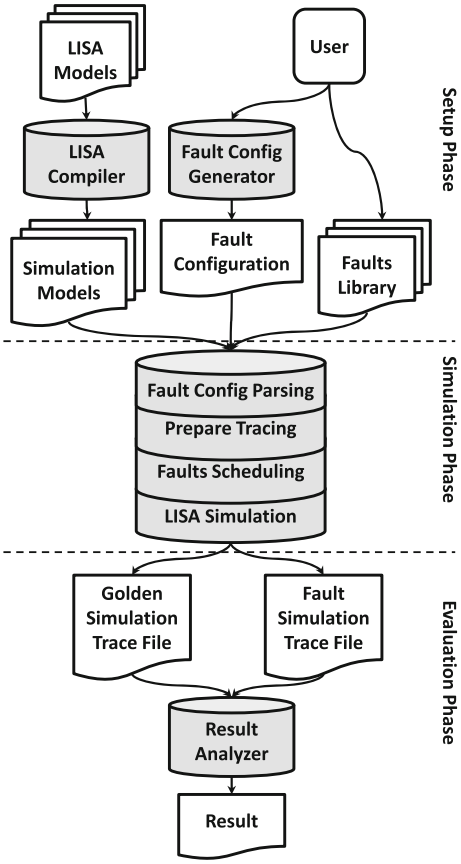


Fig. 4.1 LISA-based fault injection and evaluation flow [215] Copyright ©2013 **IEEE**

4.1.2 Flow of LISA-Based Fault Injection

The detailed flow of LISA-based fault injection is described in this section. The overview of the flow is present in Fig. 4.1, which consists of three phases in the following.

4.1.2.1 The Setup Phase

This phase generates the processor simulation model supporting fault injection and configuration. Cycle-accurate processor simulation model is generated from the LISA descriptions, where all global hardware resources such as registers, global signals, memories, and pins are subjected to fault injection. In terms of hybrid fault injection, global disturbance signals are declared to modify the original logic contents inside LISA operations. Figure 4.2 uses an example to show the code-modification

```
OPERATION add IN pipe.EX
{
 BEHAVIOR
  {
   alu_out = alu_in1 + alu_in2;
  }
 ACTIVATION
  {  write_back   }
}
```

Without Fault Injection

```
#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}
}
```

With Fault Injection

Fig. 4.2 Fault injection through disturbance signals in LISA operation [215] Copyright ©2013 **IEEE**

based method to inject the fault into an operation in the pipeline stage EX of an RISC processor. Explicitly, all the read identifiers in the behavioral statement and activation condition can be masked through bitwise operators with the disturbance signals, whose values can be assigned dynamically during simulation through simulator interfaces. Different fault types are achieved through pragmas by different bitwise operators. For instance, Fig. 4.2 presents how to realize bit-flip fault on signals *alu_in1* and *alu_in2*, as well as the condition to activate operation *write_back*.

Both the global resources and disturbance signals can be modified dynamically through simulator commands. To ease fault injection during execution, a fault configuration file which describes the fault properties are parsed to the simulator to schedule and inject the faults. A user friendly graphical interface has been developed to generate the configuration file in XML format. A snapshot of the graphical interface is shown in Fig. 4.3. Several configurable fault properties are listed in Table 4.2.

The injection time and durations are based on the unit of clock cycles which is the minimal time unit of cycle-accurate simulation. Several properties of faults are specified in a range between the maximal and minimal values, with an annotated

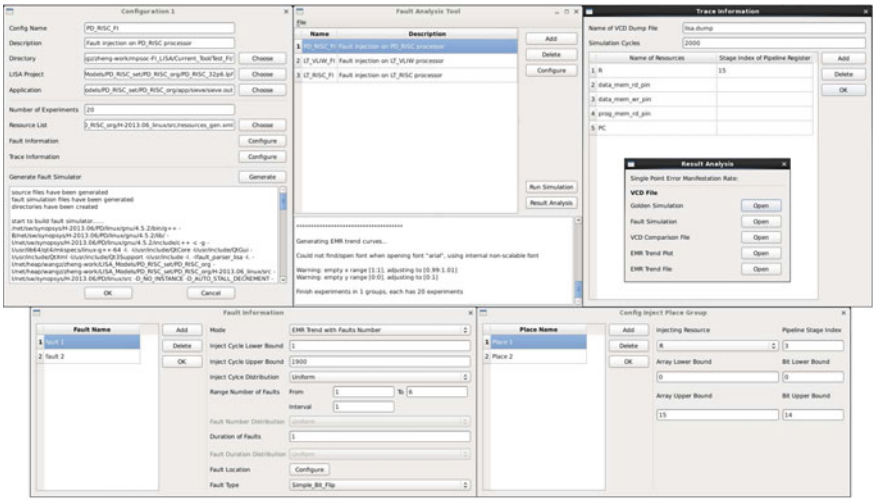


Fig. 4.3 Graphical user interface for fault configuration and evaluation

Table 4.2 Fault properties in configuration file [215] Copyright ©2013 **IEEE**

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 (name of resources)
	Bits range of faulty resources
	Array range of faulty resources
	Count range of injected faults
	Probability distribution of range based fault properties
	Bit Error Rate of faults
Fault Evaluation	Simulation time (unit clock cycles)
	Total number of experiments
	List of tracing resources

probability distribution function. The purpose is to support the range based faults which are determined statistically under specific probability distributions. The resolution of faulty location is fine-grained into a specific bit. Each bit of configured fault resource is annotated with Bit Error Rate (BER). The BER can be used to construct physical faults measured from taped-out circuits or circuits-level statistical Monte Carlo simulations. Provided that no faulty locations are specified, the tool random select locations from all processor resources. The fault types are selected from the list

in Table 4.1. Extension to the fault types can be constructed through adding inherited data structure in the fault library.

Other than fault configuration, the methods for error evaluation is also listed in Table 4.2 supporting customized definition on error conditions. By executing a given number of experiments where each runs a given cycles, the simulator traces the errors on given resources to compute the *EMR* value. The list of tracing resources provides an easy interface to compare signal values in the simulation *Value Change Dump*(VCD) file. Listing 4.1 provides an example of XML file for fault configuration and error injection.

The fault injection supports generic LISA models and the software applications, which are also configured through the graphical interface.

4.1.2.2 The Simulation Phase

This phase injects the configured fault during behavioural simulation. Applying the methods of programming interface from Synopsys Processor Designer [184] user can update the resource values conveniently. Several methods for model initialization, single step execution, resource tracing, acquiring and assigning are used for fault injection. The simulation phase is composed of following two steps.

1. Data structures on faults are constructed based on fault configuration file. Each fault is allocated with a fault object to store its properties. All such objects are appended onto a waiting fault list, which is sorted in ascending order with the injection time. Another active fault list is initialized as an empty list. The traced resources are added to the tracing list of processor simulator.
2. When the simulation starts, a fault scheduler is triggered to inject fault according to the injection time of fault objects in the wait fault list. When a fault starts to be injected, its data object is moved from the waiting list to the active list. Based on the remaining time of fault duration, the simulator decides whether to remove the fault from the active list. The fault injection is completed when both lists are empty. The simulator finishes the remaining simulation cycles.

4.1.2.3 The Evaluation Phase

This phase detects errors on the traced resources and performs the analysis. To compute the *EMR* value, the toggling information on the traced resources are recorded into VCD file for each experiment. The cycle wise resource values are benchmarked with the one from an error free golden simulation. Once upon detected value mismatch, detailed information on the location and time of fault injection and detected errors are recorded for further analysis. the final *EMR* value is calculated based on Eq. 4.1 after all experiments.

```
<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>  %Values on register file R are traced
    </TracedResources>
  </ConfigFaultAnalysis>
</ConfigFaultSim>
```

Listing 4.1: Example of fault configuration File in XML format

4.1.3 Timing Fault Injection

In most high-level simulation frameworks, the 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 Fig. 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 is analyzed during logic synthesis or placement and routing using Static Timing Analysis (STA) [81]. Such bitwise delay information is back-annotated as extra information for the hardware resources in an instruction-set simulator.

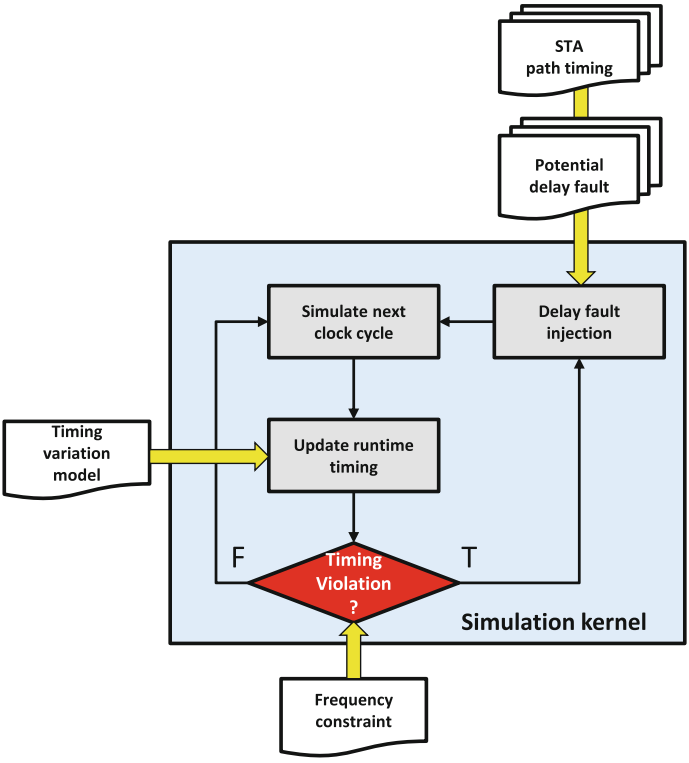


Fig. 4.4 Simulator extension for injection of delay faults

- 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

To demonstrate the effectiveness of proposed fault injection technique, two case studies are conducted in this section. First, a study on accuracy of error detection is present by benchmarking proposed technique with a state-of-the-art RTL fault injection. The second study shows the power of fault injection during architecture exploration, where a RISC processor is quickly and securely customized for cryptographic application using proposed framework.

4.1.4.1 Benchmarking with RTL Fault Injection

The Verilog-based fault injection technique in [44] is chosen for benchmarking, which is based on an extension to the commercial Verilog simulator using Verilog programming interface. Similarly, the fault simulation is achieved through scheduling the event queues of Verilog simulator. The generosity of standard Verilog language support provides fault injection into both RTL and gate-level models.

Experiments are performed on an RISC processor as well as VLIW processor from the IPs of Synopsys Processor Designer. Proposed LISA-based fault injection simulate faults directly on the instruction set simulator of the model, while Verilog fault injector performs on the automatically generated Verilog codes from the IP to ensure both models behaves exactly the same. The application running on the processor is *Sieve of Eratosthenes* which detects prime number in a given range of numbers. In the following sections, experiments on accuracy and running speeds are discussed.

Accuracy Same fault configurations are applied to both fault injectors for RISC processor. Bit-flip faults are injected into 6 Verilog modules and corresponding LISA resources/operations with randomly injected locations (resource within the module) and time instance. The count of single bit bit-flip faults ranges from 1 to 6 with a fixed fault duration of 1 clock cycle. Simulation time is set to 1,200 clock cycles which is end of the application time. Each measurement point of *EMR* value is the computed from 3,000 fault injection experiment (Fig. 4.5).

The EMR trends with the increasing count of faults are shown in Fig. 4.6. It is observable that proposed high-level fault injection achieves similar accuracy as Verilog-based fault injection under the EMR metric. The differences in absolute values for the same experiment setup come from two factors. Primarily, although with the same fault configuration is provided, both frameworks generate their detailed list of faults independently, which are based on different sequences of randomness. Secondarily, most experiments give slightly higher EMR value for LISA fault injection than RTL injection. This is caused by the fact that signals which have minor effects on the logic behaviors are only present in the generated RTL codes instead of the LISA description. Faults injected on those RTL signals lower the average EMR values.

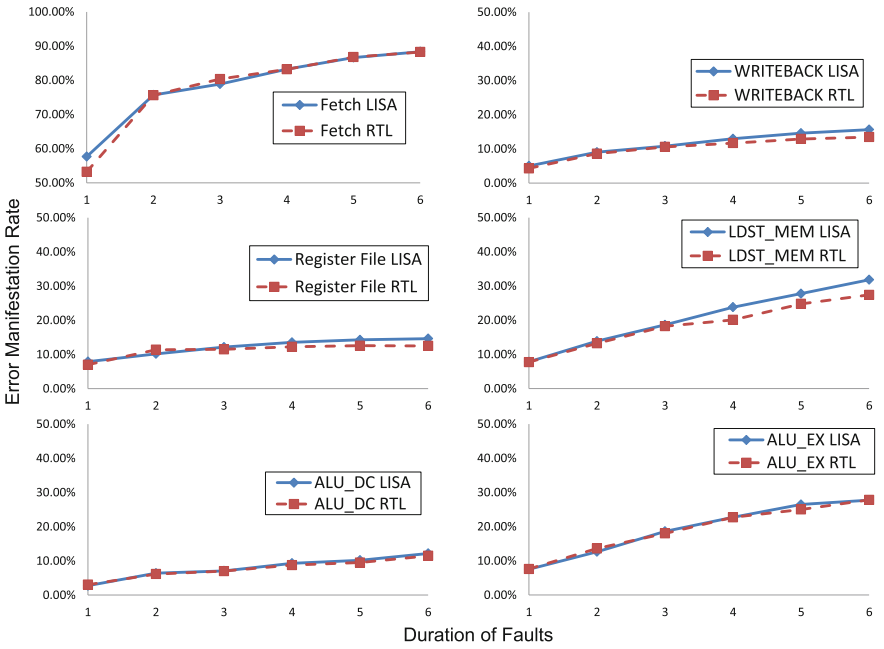


Fig. 4.5 Exemplary EMR with increasing duration of fault (RISC) [215] Copyright ©2013 **IEEE**

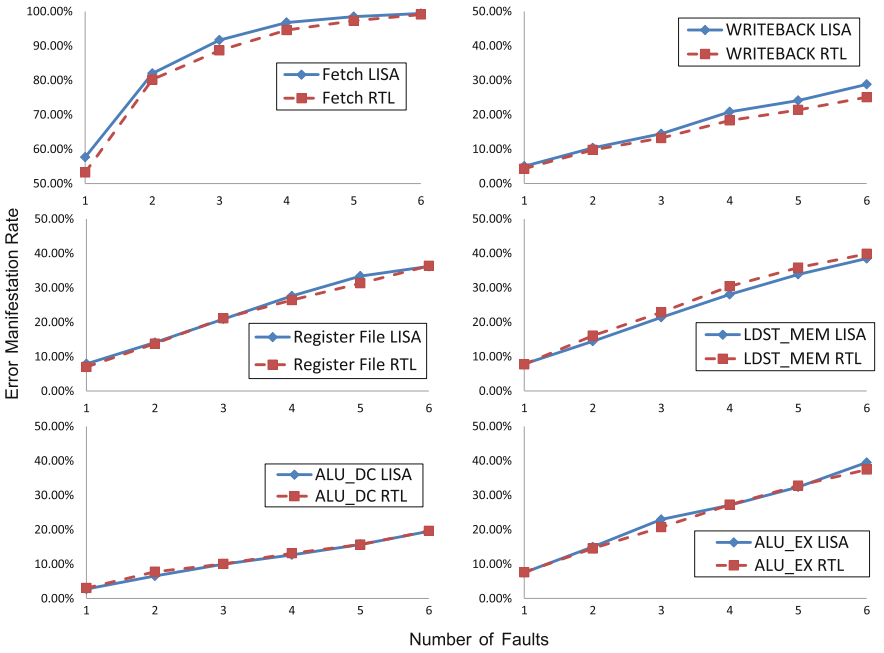


Fig. 4.6 Exemplary EMR with increasing count of fault (RISC) [215] Copyright ©2013 **IEEE**

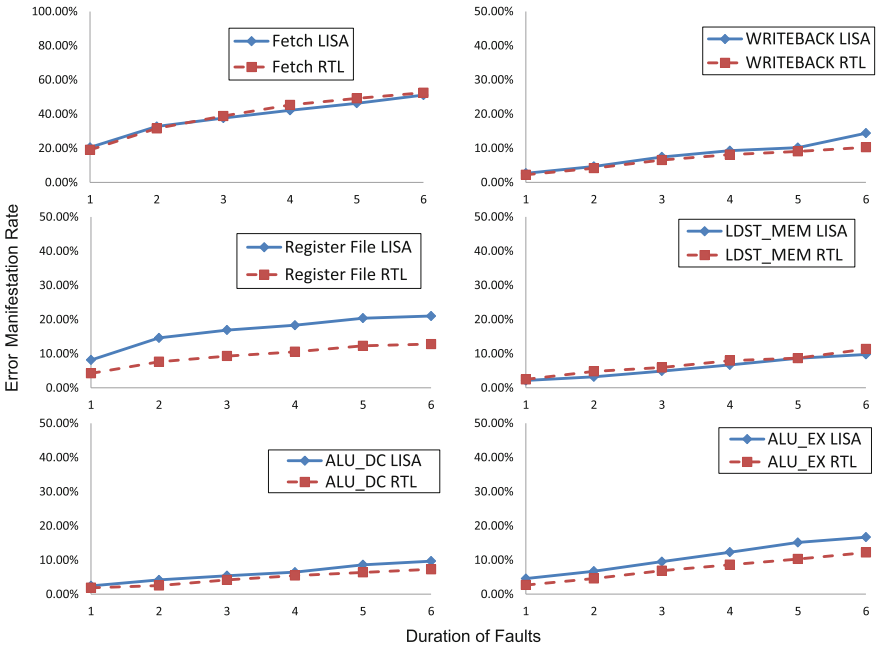


Fig. 4.7 Exemplary EMR with increasing duration of fault (VLIW) [215] Copyright ©2013 **IEEE**

EMR increases faster with fault count than fault duration. This happens since the effect of increased number of faults is accumulated since such faults are independent of each other. However, same fault with long duration affects only the same resource in adjacent clock cycles, which has a minor impact.

Compared among different hardware modules, Fetch unit is the most vulnerable among all modules, since a fault inside it can potentially affect all instructions. On the contrary, modules such as *write_back* and *alu_dc* give very low EMR values since only a few instructions directly use the outputs of such modules. This leads to the importance of architecture reliability analysis, which is discussed in Chap. 5.

Further experiments are performed to evaluate the impact of faults on different architectures. Another VLIW processor from Synopsys IPs is investigated. Same fault configuration on the modules of RISC processor is applied to the VLIW processor running the same application. Figure 4.7 presents the trend of EMR with fault duration for VLIW. The results give similar EMR trends and accuracy with Verilog fault simulation. Furthermore, EMR values under same fault configuration are smaller in most VLIW modules than the RISC ones. One reason is that VLIW processor has 4 parallel instruction slots. It is often that only a few of the slots are occupied with meaningful instructions. Consequently, random fault injection into the idle datapaths will greatly reduce the average EMR values. This effect does not happen on register file since both architectures have the same number of general purpose registers. The mismatch of LISA-level EMR for register file module compared with the one from

Table 4.3 Benchmark of fault simulation speed [215] Copyright ©2013 **IEEE**

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

Verilog fault injection is caused by the huge number of access ports generated for Verilog model of VLIW. The faults which are modeled as register value changes in LISA model is not equivalent to the value changes on ports in Verilog model. For RISC processor such gap is smaller since only one set of access ports exist for the RISC pipeline.

Speed Speed is another major metric in evaluation of any simulation framework. Table 4.3 shows the execution time required to finish 3,000 experiments by both frameworks. Both simulations are performed on the same host machine.

It is easily seen that LISA-based fault simulation has as a factor of 10x speeding-up than the Verilog-based one. The reason is that RTL simulators detect the next event of simulation dynamically based on the value changes in the sensitivity list of logic blocks, whereas simulation in instruction-set level schedules operations statically according to activation conditions. More on discussions of cross-layer simulation speed is referred in [112].

4.1.4.2 Exploring Reliability Using Fault Injection

The proposed fault injection under an ADL framework enables fast reliability exploration through both software and hardware customizations. This section demonstrates such efficiency by protecting a RISC processor from fault injection attack on Advance Encryption Standard (AES) [45] application.

The implementation of AES application from Brian [26] is compiled by the C compiler generated from Processor Designer, which is executing on the RISC processor. The bit-level fault injection attack from [176] is adopted, where selected bits of the temporary cipher as the beginning of the last encryption round are flipped to acquire the cipher key. Using such method, the 128-bit AES key is obtained with less than 50 faulty ciphertexts. The following section intents to realize such attack and provides an extra protection mechanism to the RISC processor.

Vulnerable resources identification To identify the vulnerable resources for storing the temporal cipher between 9th and 10th encryption round, fault free simulations are executed in assembly level to address the locations of storage. It is found that the temporal results are kept in four general proposed registers R[5], R[6], R[10] and R[11].

Software and architecture exploration Software technique to enhance application security can be implemented directly by modifying the source code in C language. By

Table 4.4 Synthesis Result for Protected/Unprotected Designs [215] Copyright ©2013 **IEEE**

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

executing the last two encryption rounds twice and compared their results to detect if an attack has occurred. The simulation shows that such method increases the total executing time by 1%. However, as detected from the simulation that the repeated execution also stores the temporal cipher in the same registers as the original one, it is highly probable that a fault attack lasting for a longer period can effect repetitive rounds, resulting in undetectable errors.

Alternatively, hardware based modular redundancy approach is introduced by duplicating the registers storing the cipher codes. 4 additional registers are declared to be used in the logic operations. All logic modules reading from/writing to the original registers also read from/write to the alternative ones. A comparison is performed before value read to detect potential errors, which will halt the processor once upon error detection. Only a few minutes in architecture customization is spent for aforementioned modification. The new processor simulator and Verilog descriptions are generated automatically. Other hardware oriented techniques can be also implemented in such ADL framework easily.

Fault simulation Same faults configuration are applied to inject faults in the original registers storing error-prone ciphers. Fault injection results show that mismatches happen on the original registers with protected ones, which indicate a successful error detection.

Hardware implementation The architecture is synthesized with faraday 90 nm technology library [61] by Synopsys Design Compiler [182]. The result in Table 4.4 gives the estimates on physical factors of the design, which shows a 3% increment in critical path timing, 43% in area and 30% in power consumption compared to the unprotected architecture. The extra area is caused by the protection registers (contributed 18.5% of area increment) and error detection logic (contributed 24.5%). Such results review the trade-off between security and hardware cost.

4.1.5 Summary

A high-level fault injection technique for commercial processor design environment is present in this work. Both code-modification and simulator-command methods are applied to increase fault coverage. Benchmarking with RTL fault injection framework demonstrates the accuracy and simulation speed. Hardware oriented reliability exploration performed at ADL abstraction facilitates fast prototyping of fault tolerant architectures.

4.2 System-Level Fault Injection

The growing complexity and performance requirement of applications give rise to the architecture solution of Multi-Processor System-on-Chip (MPSoC). Designing of the multi-processor system has a stronger requirement on systematic high-level design infrastructure beyond traditional RTL-based design. To this end, SystemC has been integrated into the standard design flow for complex SoC modeling. It takes advantage of a library of C++ classes and functions supporting concurrent simulation of processes and threads. Several platform-level design tools come into play by Electronic System Level (ESL) tool vendors. This directly influences the reliability research community that reliability exploration techniques, such as fault injection, must be coped with the system-level design. Even though processor specific fault injection techniques have been widely investigated [35, 153], system-level fault simulation technique is relatively less explored [122].

Contribution In this section, an efficient and generic fault injection technique is developed for SoC components, such as processor IPs, memory, and bus. Such tool facilitates system-level reliability exploration. For processor cores such ass ARM9 [9], fault injection is realized through the inherited programming interface of C++ components of abstract processor models as the technique in Sect. 4.1. Regarding memory and buses, new fault injection methods have been developed to achieve cycle-accurate fault injection.

4.2.1 Fault Injection for System Modules

A snapshot of the commercial system-level design environment from Synopsys Platform Architect [183] is present in Fig. 4.8. The developed fault injection techniques target different modules including the processing elements, but and memories, which are introduced in the following.

4.2.1.1 Processor Fault Injection

Processor fault injection takes directly the technique developed in Sect. 4.1 based on architecture description language LISA, where the LISA Application Programming Interface (API) is adopted for modifying the execution states. The IP cores used in Platform Architect inherited the methods from LISA simulation models by constructing a SystemC wrapper around the API functions of cycle-accurate processor simulators. This ensures the fault injection with similar accuracy from register transfer level.

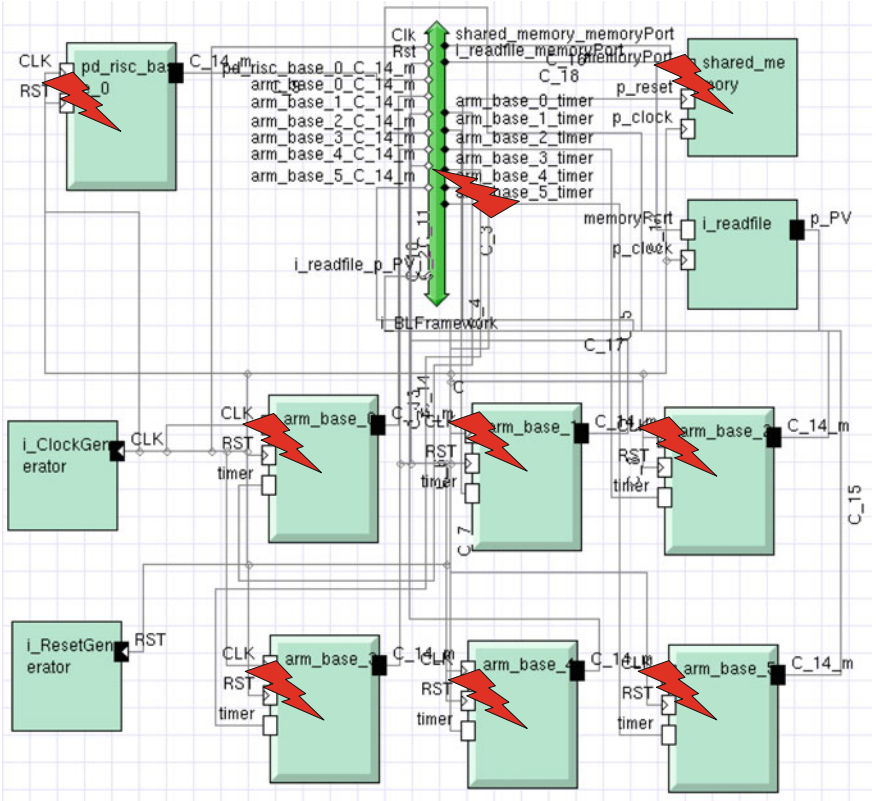


Fig. 4.8 System-level fault injection on virtual prototype [201] Copyright ©2014 **ACM**

4.2.1.2 Bus Fault Injection

Modeling faults in the bus like components is considerably simpler due to its limited states during communication. In SystemC realization of data transmission, the data is essentially an argument of the transmission function. Such data is subjected to fault injection by directly modifying its value at a specific clock cycle. Similar to processor cores, the initialization phase of fault injection parses the faults from a configuration file and schedules them in a queue like structures according to the time of injection. Once upon data transfer function is called, the injector detects whether the current time instance coincides with the time of fault injection to actually inject the faults.

4.2.1.3 Memory Fault Injection

Other than processors and buses, memories in SystemC usually do not need to be clock-sensitive under behavioral simulation, which is required by cycle-accurate fault injection. To address this, an extra clock-sensitive SystemC method is introduced to maintain a clock counter for each memory block. Such counter is continuously checked with the fault injection time. The proposed method can be applied to other clock-insensitive SystemC modules. For the configuration of memory faults, the user is required to provide the array indexes.

4.2.2 Experimental Results

The Operating System Application Specific Instruction-set Processor (OSIP) [28] implements a dedicated hardware accelerator of kernel functions in the operating system, especially task scheduling and synchronization in heterogeneous MPSoC. In this work, an MPSoC using OSIP for scheduling is adopted to demonstrate the effects of fault injection. The platform is composed of 7 ARM926EJ-S processors as processing elements (PEs) and one OSIP core. Two applications have been investigated.

4.2.2.1 H.264 Decoder

The application decodes input data into the video stream. PEs dynamically get task assigned from the OSIP. Faults are randomly configured into PEs while the OSIP is set as fault free. Figure 4.9 presents several effects of application errors resulted from the faults, which are exhibited as pixel error, thread error and fail to process.

4.2.2.2 Median Filter

OSIP can accelerate applications in image processor by scheduling independent tasks of image pixels onto multiple PEs. For instance, median filter reduces image noise by taking the median value of adjacent image pixels. Figure 4.10a and b presents the original image with noise and the one after filtering. A fault-tolerant implementation of median filter schedules additional tasks to other PEs whenever one PE is detected to be unresponsive. Watchdog timers are implemented to reboot the unresponsive PEs, which get new tasks from OSIP. The watchdog time is set to be $3\times$ of the regular processing time for one data output. Figure 4.11 shows median filter under several fault configurations. The first experiment is error free, which indicates no difference in running time with or without timers. For the rest experiments, 100 bit-flip faults are injected on each specified PE. Without watchdog timers, the timing for image processing grows significantly as the number of unresponsive PEs. However,

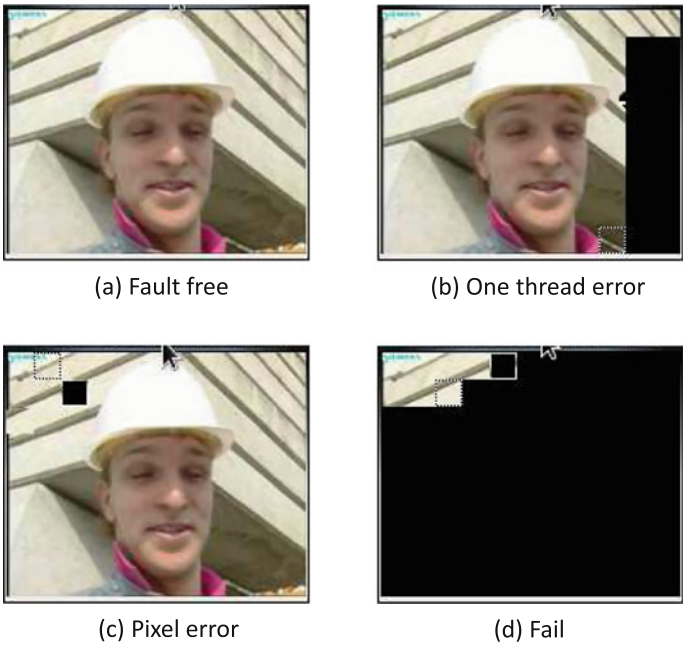


Fig. 4.9 H.264 decoder with fault injection [201] Copyright ©2014 **ACM**

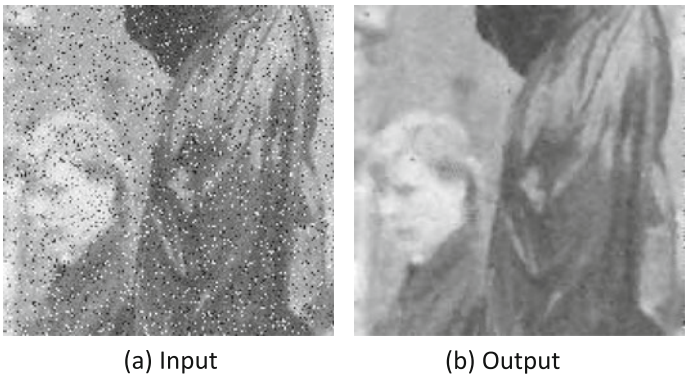


Fig. 4.10 Median filter: original and filtered image [201] Copyright ©2014 **ACM**

only slight overheads are occurring for the PE with watchdogs. This happens since tasks are re-assigned by the OSIP when watchdog reboot the failing PEs. The system-level fault injection experiments prove the effectiveness of fault tolerant technique by using watchdog timers as error detection and correction mechanism.

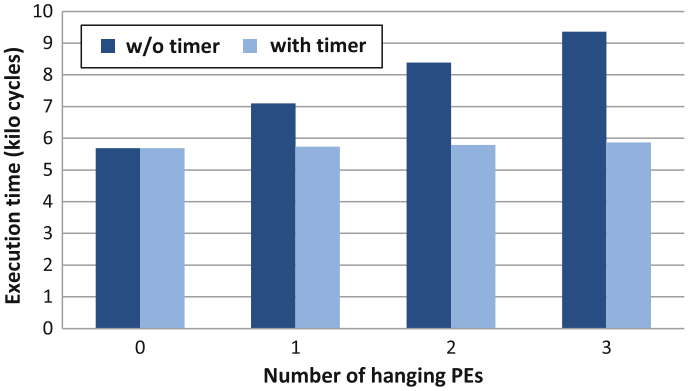


Fig. 4.11 Median filter: reliability exploration [201] Copyright ©2014 **ACM**

4.2.3 Summary

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

4.3 Statistical Fault Injection for Impact Evaluation of Application Performances

Shrinking transistor sizes and the need to reduce pessimistic[1] design margins renders the circuits with increasingly reported timing errors caused by parametric variations and noise on the supply voltage. The propagation of such timing errors through the architecture results in application failures. To prevent catastrophic impacts, approaches have been developed to either predict the errors and applying voltage/timing guard bands at design time or detect the errors at run time and performs error correction [25, 59]. Such techniques incur large power and performance penalties.

Alternatively, approximate computing paradigm tends to accept the errors and trade-off power against application qualities. This is heavily based on the error-resilient nature of various applications [34]. However, the impact of errors on physical devices is hard to be evaluated based on existing simulation models. Inaccurate

[1]This work is collaborated with Telecommunication Circuits Laboratory (TCL) in EPFL, Switzerland and Institute of Electronics, Communications and Information Technology (ECIT) in QUB, UK. Special thanks to Dr. Jeremy Constantin, Dr. Georgios Karakonstantis and Prof. Andreas Peter Burg.

prediction of errors at design time results in inefficient techniques and possible to give a complete failure design. Therefore, an approach which integrates the effect of timing errors caused by variations into fault injection-based micro-architecture simulation framework is of high demand.

Contribution In this work, a novel approach which models gate-level timing errors during high-level instruction set simulation is proposed, which is based on the accurate characterization of the statistical nature of the timing of an open-source processor. Following contributions are involved:

- The proposed approach extends the work in Sect. 4.1, where either purely random or user configurable faults are injected, to support timing error by extracting the characterization of logic timing from a post place and route netlist.
- The accuracy of characterized timing errors is improved using gate level dynamic timing analysis (DTA).
- The initially fixed characterization under DTA for different operating conditions is extended to model the dynamic impact of supply voltage noise, which is one of the most critical timing uncertainties.
- Cycle accurate instruction-set simulation supporting characterized timing errors is used to investigate impacts of timing errors at the application layer. Such impact is quantified by output quality, energy, and performance.
- The proposed technique is applied to a 32-bit 6-stage OpenRISC core in 28 nm CMOS technology running various application kernels, which are assessed by output quality and point of first failure (PoFF).

Overall, the proposed technique does not only offer an accurate evaluation of timing errors on application performance but also assist in identifying bottleneck of hardware implementations and determining the timing margins to achieve the desired quality.

4.3.1 Setup and Case Study

In this section, we first introduce the hardware and software environment before describing the modeling approach of timing errors.

4.3.1.1 Hardware Processor Core

The 32-bit general purpose OpenRISC processor [107] is used as the target architecture. The architecture consists of 6 pipeline stages, which issues single instruction per cycle. Hardware multiplier is supported to realize multiplication with single cycle latency. SRAM with single cycle latency is used to implement both program and data memories.

During logic synthesis the constraint strategy in [38] is applied, which ensures that control paths, which are not on the critical paths, will not be immediately affected by frequency-over-scaling. This incurs slightly area and power overheads (≈5–13%) [38], however enabling a graceful path timing degradation beyond the limit caused by static timing analysis (STA). Especially, such strategy ensures only the data path of ALU units limit the maximal frequency (707 MHz at 0.7 V), while other paths are short enough when operating at a higher frequency (1.15 GHz at 0.7 V). Consequently, the modeling of timing errors under such optimization can be focused in the 32 ALU endpoint flip-flops on the critical path, while other non-ALU instructions are always safe against timing errors.

Dynamic timing analysis proposed in [38] is used to characterize the path timing for a post layout test chip, which has been fabricated by 28 nm FD-SOI CMOS technology.

4.3.1.2 Instruction Set Simulator with FI

The cycle-accurate instruction set simulator is generated from a custom LISA model of OpenRISC architecture. The simulator is enhanced by the fault injection framework in Sect. 4.1, which allows fast injection of faults into pipeline registers. The annotated injection probability on each bit of flip-flops enables the modeling of timing error under any statistical distribution.

During execution of application benchmarks, the fault is only injected into the kernel part of the application other than the booting part. Such constraint focuses the analysis on the benchmark. Timing faults tend to disturb program execution flow, usually due to the error in branch address calculation. In most case, the wrong branch will lead the processor into a deadlock state. We detect such deadlock situation by checking the value of instruction register of the processor. In case that among 100 continuous clock cycles the processor executes the same instruction, the simulation is forced to terminate earlier than the complete simulation time with a no response error report.

4.3.1.3 Software Benchmarks

The application performance is characterized by four kernels which are shown in Table 4.5 as well as their properties. Some are heavy in computation while others are rich in control flow. The performance reported in this work are based on Monte Carlo simulation. Each data point is evaluated by at least 100 simulation experiments.

Table 4.5 Overview of benchmark properties [37] Copyright ©2016 **ACM**

bench-mark	**median**	**matrix mult. (8- & 16-bit)**	**k-means clustering**	**Dijkstra**
type	sorting	arithmetic	data mining	graph search
compute	-	++	+	-
control	+	-	+	++
size	129 values	16x16 matr.	8 points (2D)	10 nodes
cycles	216 k	60 k	351 k	984 k
output error	relative difference	mean squared error (MSE)	cluster membership	mismatch in min. distance

Table 4.6 Overview of timing error models and features [37] Copyright ©2016 **ACM**

model	*fault injection technique*	*timing data*	*multi-V_{dd}*	*V_{dd} noise*	*gate-level aware*	*instruction aware*
A	fixed probability	none	no	no	no	no
B	fixed period violation	STA	yes	no	partially	no
B+	modulated period violation	STA	yes	yes	partially	no
C	probabilistic period violation (using CDFs)	DTA	yes	yes	yes	yes

4.3.2 Modeling of Timing Errors

Modeling of timing errors in the instruction-set simulator can be performed with different levels of detail. Table 4.6 gives an overview of modeling approaches and their features used in this work. Purely random fault injection is labeled as model A, which is the start point to show its limitations. Next, model B incorporates static timing analysis under given operating conditions, which resembles more real circuit's operation. The model B is further refined by considering the impact of supply voltage noise on the timing behavior, which is referred as model B+. Finally, the novel model C is introduced, which further improves the accuracy of the characterization with an even more detailed fault injection that also takes into account the dynamic timing statistics from individual instructions.

4.3.2.1 Fixed Probability

Model A resembles the situation that random bit-flips are injected into physical registers of the processor core. Each bit-flip has a fixed fault injection probability. This fault type is motivated to realize the SEU types of faults where injected faults are independent of each other. The accuracy study of this fault type has been investigated in [35].

Proved with its simplicity, model A fails to motivate itself from the physical perspective, since it neglects the timing errors which tends to occur at the end points of flip-flops on the critical and near-critical paths. Furthermore, the model does not link to any hardware operating conditions, which can not be adopted to characterize frequency and voltage impact to the applications.

4.3.2.2 Static Timing Based FI

This model relates closer to underlying hardware by annotating endpoint flip-flops with the worst case path delays generated by STA. Such delay information can be generated for a post layout netlist under different operating conditions of the technology libraries. A fault is injected to the pipeline register when the running clock period is smaller than the critical path, regardless of the instruction executing in that pipeline stage.

The issue with model B lies in its overly pessimistic estimation for real path delay. It neither considers the influence of executing instruction nor the operand values being calculated. For instance, it is extremely less probable that a timing error happens in ALU endpoint flip-flops when NOP instruction is executing on such pipeline stage. Besides, this mode does not consider the timing variation caused by supply voltage noise. Consequently, this mode cannot explore the dynamic effect of the timing errors.

To explain the over pessimism effects of this model, median filter benchmark is running on the processor under different frequencies in Fig. 4.12a. It is observed that the fault injection rate immediately rises as soon as clock period exceeds the static timing. This is because any instruction produces an error in the ALU output pipeline register. The high error rate leads to the sudden drop of application finishing and correct rates with no transition region. This is no randomness in the error model so that Monte Carlo simulation will not show any statistic information in this model. Even though present here by the median filter, all other benchmarks show the same behavior.

4.3.2.3 Supply Voltage Noise

Model B is improved to B+ by accounting for the delay variation caused by noise on supply voltage. Supply voltage noise is a primary source of gate delay variation which can be resulted from many factors such as DC-DC converters, power delivery

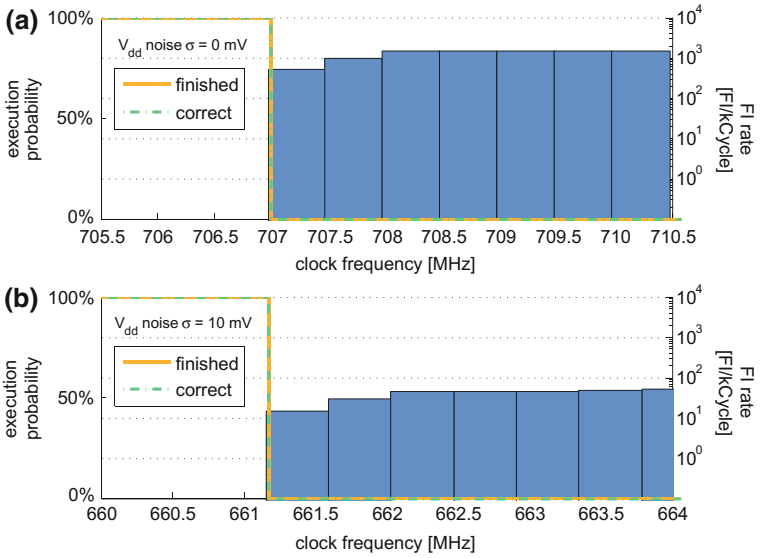


Fig. 4.12 Performance and fault injection rate of the median benchmark for **a** model B based on STA @ 0.7 V, and **b** model B+ with supply voltage noise [37] Copyright ©2016 **ACM**

networks, and circuits switching (V_{dd}-droop). In this work, we simply model the supply voltage noise by a white noise under normal distribution with a mean of 0 V and standard deviation of σ. The maximum noise value is limited to 2σ to avoid physically unrealistic voltage spikes.

Each clock cycle in the simulation generates a random voltage value based on the normal distribution. Such voltage value is translated into the variation for the path timing in that cycle. The relationship between voltage and timing is acquired from a fitted V_{dd}-delay curve, which is interpolated from the critical delay values under STA for five voltage corners (0.6–1.0 V in 100 mV steps). Such estimated value is used to approximate small delay changes around an accurate operating point. The simulator uses such technique to determine whether timing errors should be injected.

The injected error rate of model B+ is present in Fig. 4.12b for $\sigma = 10$ mV (maximum V_{dd} noise of ± 20 mV). Compared with model B, the clock frequency where errors begin to be injected is significantly lower. Higher noise σ gives smaller frequency of first error (at 661 and 588 MHz for $\sigma = 10$ mV and $\sigma = 25$ mV respectively). On the other hand, the error rate at the first frequency of error is considerately smaller (10 faults per 1000 cycles) caused by the randomness in voltage noise.

With regard to application performance, the same hard threshold between 100% successful experiment and complete failure still exists. This happens due to the fact that no instruction and data dependent impacts on error rates are modeled for path delays.

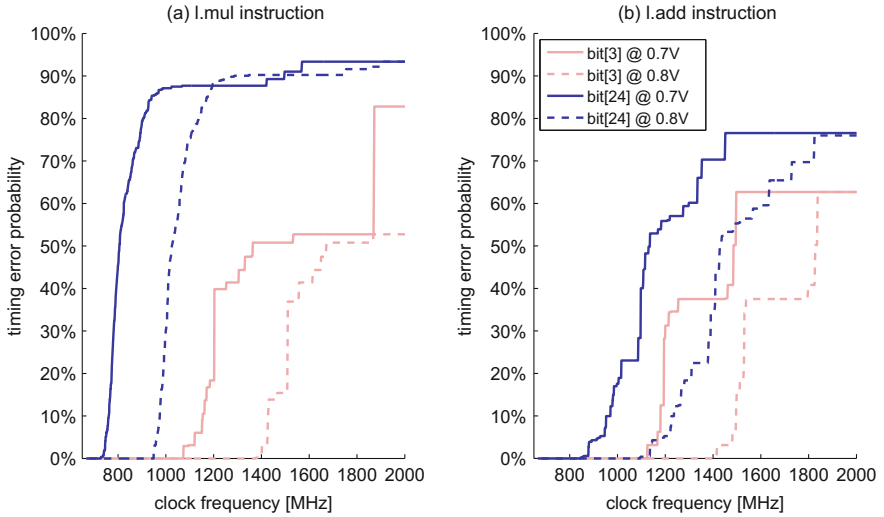


Fig. 4.13 Cumulative distribution functions of timing error probabilities extracted by DTA, for different ALU endpoints and supply voltages [37] Copyright ©2016 **ACM**

4.3.2.4 Proposed Dynamic Timing Statistical FI

To further improve the modeling inaccuracy, a statistical model C is designed to cope with instruction and data-dependent delay uncertainties. To this end, dynamic timing analysis (DTA) [36] is employed to extract accurate arriving time of last data event on all endpoint of relevant flip-flops, which is efficiently performed during the simulation of the post-layout netlist. Different instructions provide different data arriving time in the same pipeline stage. A testbench is designed to record the dynamic logic delay for 8 kCycles which cover all ALU instructions with random operands. Furthermore, the non-ALU instructions are verified to be non-critical with sufficient timing margin.

The extracted instruction-aware statistics of dynamic timing slack is applied to calculate the probabilities $P_{E,V,I}(f)$ of an endpoint E to be injected a timing error at a given frequency f. The core is supplied with voltage V, while the instruction I is executed in the pipeline stage associated with the endpoint E. We have

$$P_{E,V,I}(f) = v_f / n_I \tag{4.2}$$

where n_I is the total number of clock cycles where instruction I is encountered in the recorded dynamic timing file, and v_f is the count of cycles for which the dynamic path delay to E (including the setup time) is larger than the clock period $1/f$, i.e. cycles for timing violation on the endpoint by instruction I. The cumulative distribution functions (CDFs) for the probabilities of dynamic timing error are realized

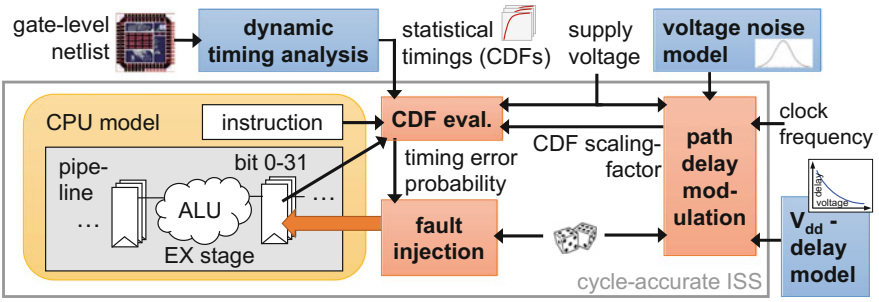


Fig. 4.14 Simulation with statistical FI (model C) [37] Copyright ©2016 **ACM**

by sweeping the frequency f. This is shown in Fig. 4.13 for two instructions on two endpoints with two supply voltages.

Figure 4.13 indicates that the multiplication instruction starts to fail at a lower frequency than the addition under the same supply voltage and ALU endpoint, which is caused by the longer delay in the multiplier. It is also observed that bits with higher significance fail at a lower frequency than bits with lower significance. A higher supply voltage shifts the CDF curve to the right. All such observations can be correctly explained by the physical characteristics of the difference in lengths of critical path and relationship between voltage and logic delay.

Figure 4.14 illustrates the overall flow of DTA based statistical fault injection of model C, which integrates the instruction-aware statistical dynamic timing information from the DTA in form of CDFs and combines it with the supply voltage noise based error modeling in Sect. 4.3.2.3. Note that the variation on clock cycle resulted from the voltage noise is used to update the CDF for each clock cycle, which gives the corresponding error probability. Other than voltage noise, the proposed approach is also able to model the effect of process variation, temperature shift and aging on the dynamic timing. Similar as the delay characterization for different voltage corner, the effect of other parameters can be evaluated by extract timing from netlist synthesized by various process corners.

Taking advantage of LISA based processor design flow, the proposed fault injection method can be fast realized through automatic generation of simulator and RTL codes, which enables accurate reliability exploration based on real physical characteristics.

4.3.3 Experiments of Statistical FI

Several case studies by proposed statistical FI (model C) are present to investigate its provided insights during instruction and application-level reliability evaluations under different operating conditions.

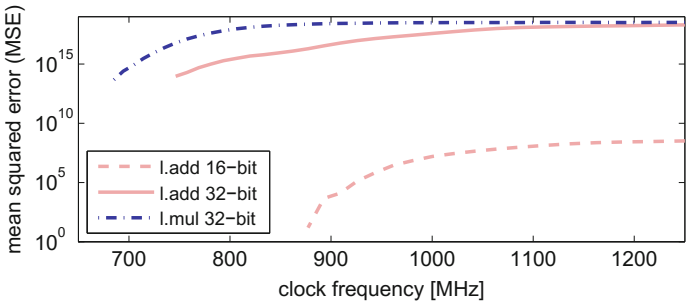


Fig. 4.15 MSE versus frequency for add. and mult. instructions at $V_{dd} = 0.7\,\text{V}$ with $\sigma = 10\,\text{mV}$ (model C) [37] Copyright ©2016 **ACM**

4.3.3.1 Instruction Characterization

To observe the different behaviors of ALU instructions, addition (l.add) and multiplication (l.mul) are tested under different frequencies. Two forms of input operands are provided for the addition, the 16 and 32-bit operands. Multiplication applies dual 16-bit operands and gives 32-bit results. Random operands are generated while the core is operated at 0.7 V with $\sigma = 10\,\text{mV}$ voltage noise. The mean squared error (MSE) is used as metric to evaluate the timing errors, which is shown in Fig. 4.15.

The plot indicates the first errors (MSE > 0) occurring at 877, 746, and 685 MHz for 16-bit addition, 32-bit addition and multiplication respectively. The large difference of the point of the first failure (PoFF) among different ALU instructions proves the importance of timing error modeling across instructions. The difference in PoFF for the same instruction (16 and 32-bit additions) also shows the significance of modeling on bit-level granularity. Besides, all instructions tend to saturate the MSE at a maximal value for increased frequency.

4.3.3.2 Impact of Frequency, Voltage, and Noise

The proposed fault injection captures details of underlining gate-level implementation which facilitate exploration of the transitional region beginning from the frequency where errors start to appear. Such region is sensitive to frequency-/voltage-overscaling and supply voltage noise. The application level impact of these effects can be characterized by four metrics: application finishing rate, the probability of correct execution, the error injection rate and relative error of program output.

Figure 4.16 presents above mentioned metrics for the median benchmark for a range of operating frequencies. Among all sub-figures (a-f), two supply voltages are given with three levels of voltage noise. Each point of the evaluation is averaged by 300 Monte-Carlo simulations. The plots show only regions with errors when the lower frequency region resulting in no errors is labeled as “n/a”.

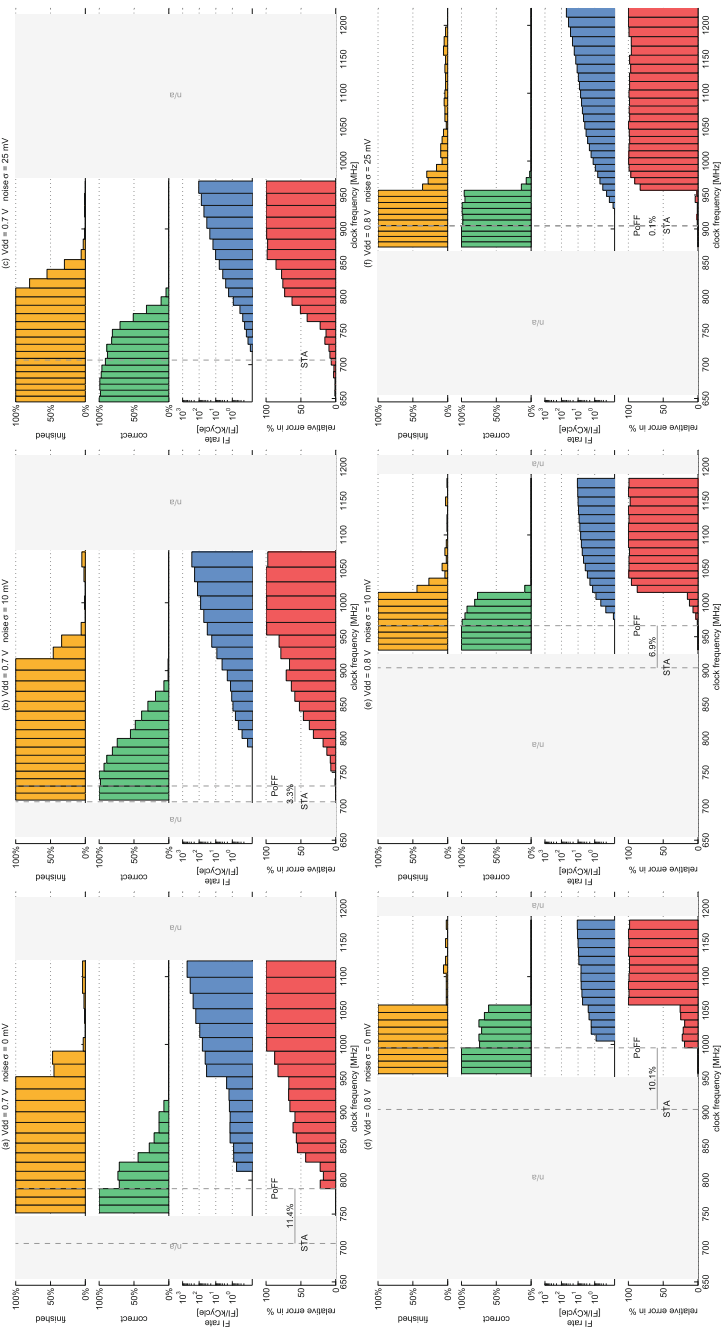


Fig. 4.16 Program performance for the median benchmark for different V_{da} and V_{da}-noise (model C) [37] Copyright ©2016 **ACM**

The pessimistic frequency limit caused by STA is also marked in the figure. Under smaller noise level, significant frequency gain is observed from the STA limit to the PoFF frequency, which provides the bandwidth for frequency-overscaling. The increment of the noise levels σ reduces the gap between STA limit and PoFF, while also smooths out the values of most metrics in the transition region, other than the relative error of output. However, the relative error is highly application dependent and tends to increase drastically from a threshold of error rates.

By increasing the supply voltage, all plots of metrics are shifted to the direction of high frequency. Higher frequency also tends to sharpen the changes in transition region hence reduce the total width of such region. This leads to an explosion of errors after PoFF for high voltage. This implies that low voltage which gives wider transition region is more favorable for approximate computing applications.

4.3.3.3 Performance Comparison of Benchmarks

Another key feature of the model C is its ability for investigating distinctions on different application kernels, which are composed of different instruction types and sequences. This section demonstrates how the applications behave differently under the same operating point of 0.7 V with a supply noise of $\sigma = 10$ mV.

The 8 and 16-bit matrix multiplication (Fig. 4.17a and b) exhibit similar application behaviour. The smaller bit width for the 8-bit version develops a higher rate of correct execution than the 16-bit one. The MSE curve scales similarly for both versions, with different absolute values caused by differences in operands and range of results.

In contrast to matrix multiplication, the K-means (Fig. 4.17c) results in an error injection rate which is one order of magnitude lower at the same operating point. This is explained by the fact that K-means kernel has significantly less timing critical multiplication instructions, which is prone be affected in the transition region. However, with the percentage of points with clustering errors to evaluate results of K-means, the kernel develops large performance degradation (30–40%) around the STA limit, even though the benchmark has a relatively higher finishing rate.

Finally, a very narrow transition region characterize the Dijkstra benchmark (Fig. 4.17d) which is simulated with higher resolution in frequency. It is observed with quite a high gain of frequency (3.5%) which does not exist for other benchmarks. Nevertheless, 4% of further frequency increase beyond the PoFF cause the complete failure of the application with a still very low FI rate (below 1 FI per kCycle). Such observation is caused by the highest amount of control instructions and least arithmetic ones out of all benchmarks. Whenever an error happens to the control flow (wrong branch address), the program completely fails directly.

Figure 4.17 also marks the frequency limit under model B+, where complete failure threshold at 661 MHz equally applies to all benchmarks. This again proves the above application-level analysis can be only provided by the model C.

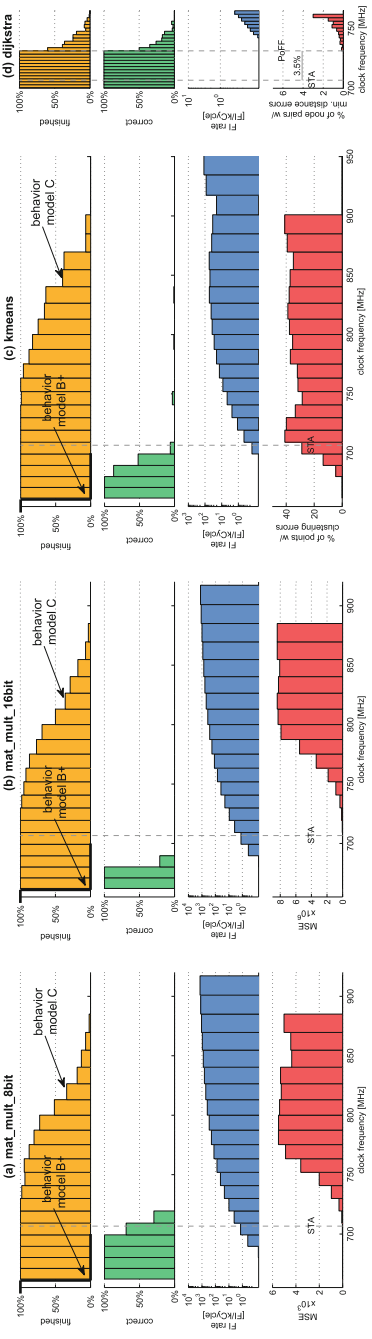


Fig. 4.17 Program performances for various benchmarks at $V_{dd} = 0.7\,\mathrm{V}$ with V_{dd}-noise $\sigma = 10\,\mathrm{mV}$ (model C) [37] Copyright ©2016 **ACM**

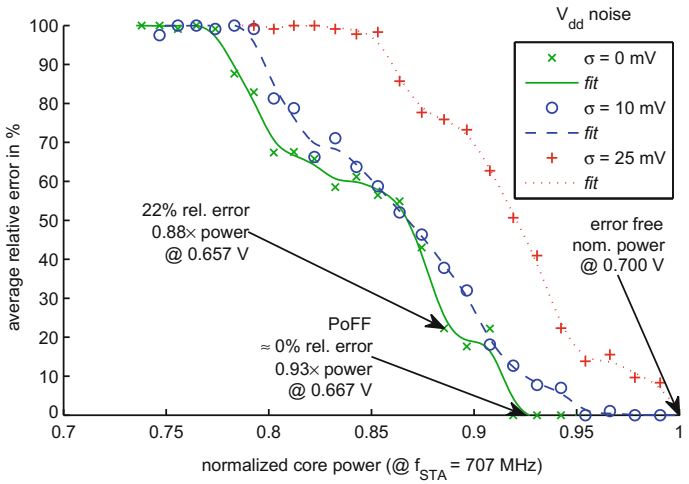


Fig. 4.18 Relative error versus core power consumption trade-off for the median benchmark (model C) [37] Copyright ©2016 **ACM**

4.3.3.4 Error Versus Power Consumption Trade-Off

One key motivation to accept error from happening within the frequency transition region is the saving on power consumption. Here we present the trade-off between power and error rates under the proposed model C.

The power saving and output quality degradation are characterized for the median benchmark. The power savings are calculated by translating the gain of frequency over-scaling into an equivalent reduction of the supply voltage. After that, the power value can be obtained from the quadratic fitted curve between two reference point under post-layout power simulation of the design. The reference points are 10.9 μW/MHz @ 0.6 V and 15.0 μW/MHz @ 0.7 V. Leakage power is ignored here which contributes less than 3%.

The relationship between MSE metric and the normalized power of median filter is presented in Fig. 4.18. The core operates at a fixed nominal frequency of 707 MHz which is the STA limit at 0.7 V. The error-free run with the normalized power of one is shown in the bottom right. The PoFF of zero voltage nose is reached at 0.93, further power reduction increases the errors where a 22% error rate is shown for 0.88 of relative power.

The impact of supply voltage noise is also considered. It is observed that noise at $\sigma = 10$ mV gives a close match of the curve to the no noise one. However, the noise at $\sigma = 25$ mV infers the PoFF with even no power gain.

4.3.4 Summary

This work models timing errors on cycle accurate instruction-set simulators which enables fast exploration on the impacts of physical characteristics, such as frequency, voltage, and noise, on application performances. Several error models are proposed in order to benchmark the accuracy of physical modeling. Demonstrated with several representative benchmarks, it is shown that the proposed statistical fault injection approach provides significant accuracy improvement. Such methods refine the error modeling by considering instruction-level timing statistics, which is obtained from dynamic timing analysis of the post-layout open source processor.

4.4 High-Level Processor Power/Thermal/Delay Joint Modeling Framework

As reliability becomes an essential factor in the design of the 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 Sect. 4.1 provides a user configurable approach to simulate processor behavior under fault. However, reliability effects, especially aging and soft errors, have a 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 the 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. [189] [190], where each instruction is provided with an individual power model. The runtime power can be determined through the profiling of executed instructions. Wattch [43] 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 [113] models all dynamic, static and short-circuit power while providing a joint modeling capability of area and timing. To increase modeling accuracy, a hybrid FLPA(functional level power analysis) and ILPA(instruction level power analysis) model [22] is elaborated which advantageously combines the lower modeling and computational efforts of an FLPA model and the higher accuracy of an ILPA model. The trade-off is further explained in [142] with a 3-D LUT and a tripartite hyper-graph.

The heat dissipation from power consumption leads to increased and unevenly distributed temperature which causes potential reliability problems [5, 98], where the research committee demands highly for architecture-level thermal management techniques. Consequently, accurate architecture level thermal modeling has received huge interests. In this domain, HotSpot [175] 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 [51, 86].

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 [62]. In the domain of digital processor design, Cacti [139] estimates power, area, and timing specifically for the memory system. McPAT [113] jointly models power, area and timing for individual system-level blocks including cores and memories. Reference [84] applies a joint performance, power and thermal simulation framework for the design of network-on-chip. Reference [187] 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 modeling of physical parameters. Previous work also lacks the ability to accurately estimate power/temperature with application-specific switching activities. The reason is that modeling and simulation are treated as separate issues, where the modeling 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 being addressed.

Contribution In this work, a joint modeling framework is demonstrated by integrating power, thermal and logic delay in a high-level processor design environment, where both accurately modeling 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 [215], where faults are modeled as delay variation on logic paths resulted from instantaneous power and thermal footprints. By automating the complete modeling 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 the following manner. Section 4.4.1 discusses the approach of high-level power modeling and estimation for LISA based processor design framework. Section 4.4.2 illustrates the thermal modeling and integration using HotSpot package. Section 4.4.3 introduces the approach of high-level delay simulation. Section 4.4.4 focuses on the automation flow and analyses its runtime overhead.

4.4.1 High-Level Power Modeling and Estimation

The runtime power consumption for LISA units (operations and resources) can be characterized from power simulation of low levels such as layout or gate-level. The power models are integrated into the instruction-set simulator to estimate power

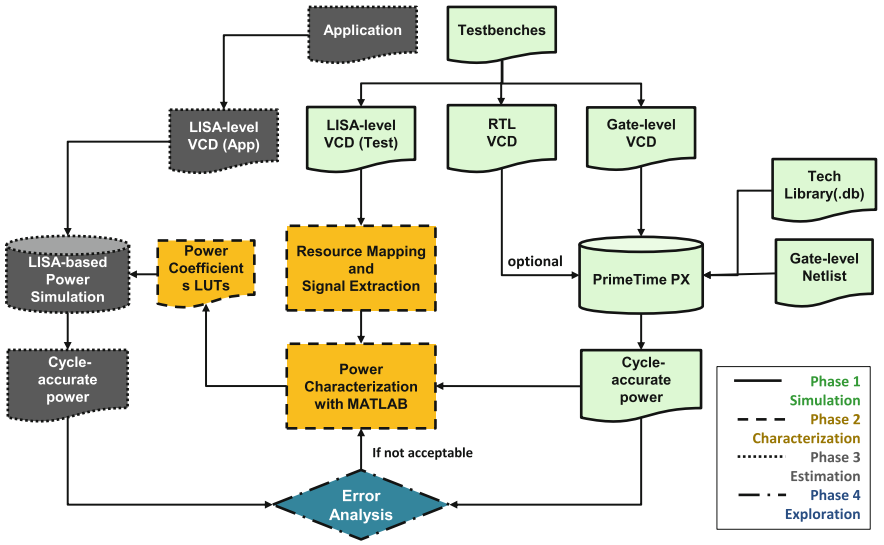


Fig. 4.19 LISA-based power modeling and simulation flow [206] Copyright ©2013 **IEEE**

with significant speed-up. The accuracy of the simulation is related with the level of details during characterization. This section explains the proposed modeling flow and features. First, an overview on the power modeling flow is presented. Second, construction of architectural power models is illustrated. Afterward, approaches to handle power related factors are introduced.

4.4.1.1 Flow Overview

Figure 4.19 illustrates the power estimation flow consisting simulation, characterization, estimation and exploration phases.

Simulation This phase collects cycle-accurate power data from low-level power simulation. The testbenches for each simulation is constructed with single instruction type with random operands, which is used to create power models for individual instruction. To improve modeling accuracy, one group of instructions e.g. ALU instruction can be further differentiated as addition, subtraction, multiplication to collect corresponding power traces due to their obvious difference in power. Besides power data, the trace of switching activity on interface signals of individual architecture unit is also gathered.

Characterization Both traces of power and switching activity are required to characterize the power coefficients for interface signals of architecture units. Multivariate regression analysis is utilized for coefficient extraction. Enhanced regression orders,

such as polynomial order instead of linear one, can be used to improve regression accuracy while increase extraction efforts.

Estimation Applications running in instruction set simulator (ISS) can calculate runtime power on each unit with extracted power coefficient. The estimated power is recorded in PrimeTime power format for benchmarking purpose.

Exploration The estimated runtime power from ISS is benchmarked with low-level power simulation to check for modeling accuracy. The estimation accuracy can be further improved by the techniques in the simulation and characterization phases.

4.4.1.2 Power Modeling

Finer power modeling from granularity of instruction to architecture units improves the accuracy of estimation. ADL-based architecture design environment facilitates the construction of power models on different granularities. Figure 4.20 lists the hierarchical representation of an RISC processor with 5 pipeline stages. The instruction level power model can be constructed by power for various instructions running on the core at level 0. Furthermore, architectural power models are created for the hardware modules at level 1 and 2 for each instruction. The proposed unit-level modeling further extends the pipeline module into operation units in level 3, which shows significant power difference among operation units. Such approach addresses the modeling accuracy through model decomposition, while also introduces a systematic modeling approach.

Figure 4.21 illustrates the concept of unit-based power model with a block representing either a logic operation or a storage unit. The power model for each unit is constructed using the cycle-wise toggling information or *Hamming Distances* of interfacing signals. The run-time power of the individual unit is estimated according to the weighted summation of the power of all interfacing signals, where the weights are the power coefficient extracted from characterization phase.

Figure 4.21 gives an example of coefficient extraction for single variable using interpolation technique [101]. Both linear and polynomial curves are shown with formulas. As explained previously, the order of the formula in proportional to the estimation accuracy. This work demonstrates power models under linear curve fitting for all units.

The power coefficient based method gives the desired accuracy only for circuits under the same activation modes. However, an inactivated logic operation consumes one order less power than the activated one provided same hamming distance. This happens because no logic switching does not propagate inside the circuit when it is inactive. With regard to storages, two modes, which are written and read, need to be separately considered, where register write can cause one order more power than register read. Such effects are included in the proposed modeling approach as shown in Fig. 4.22. According to the cycle-wise activation condition of operation units, the simulator applies either activate or inactivate power model for calculation. Register power equals the summation of its read power and writes power.

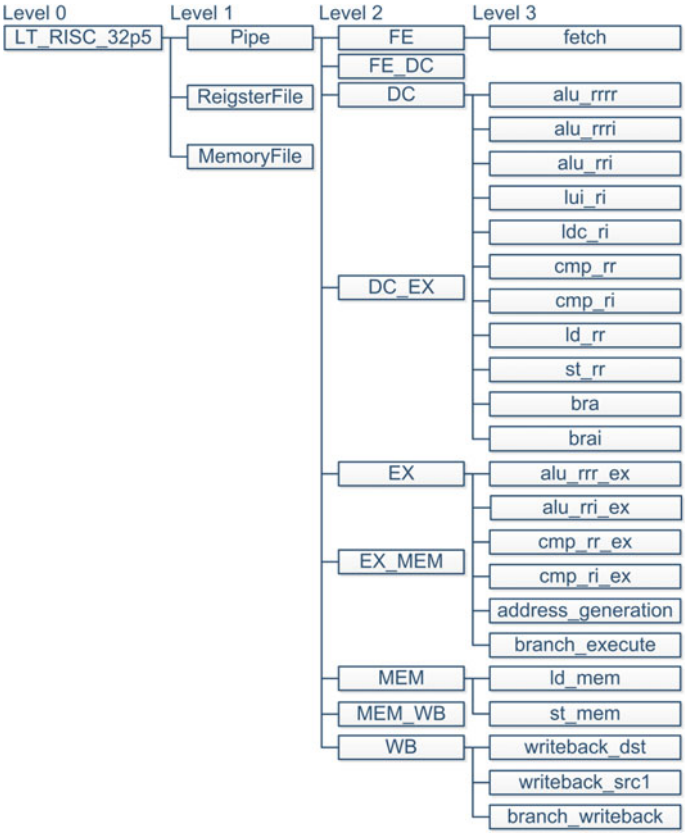


Fig. 4.20 Hierarchical representation of RISC processor architecture [206] Copyright ©2013 **IEEE**

4.4.1.3 Power Related Factors

Inter-instruction effect One key challenge for instruction level power models is the handling of inter-instruction effect (IIE), which is to quantify power caused by adjacent instruction pairs. Previous work characterizes power consumption for all instruction pairs, which demands significant efforts.

The unit level power model handles the IIE through architecture decomposition. Adjacent instructions tend to activate different architecture units in each pipeline stage, where each one has individual power models. IIE power can be considered as the deactivating power for previously active operations, which is difficult to be included for any instruction-level power models.

Custom instructions For a processor supporting N instructions in ISA, another custom instruction increases the number of instruction pairs by 2N, which is a number of pairs to be characterized under instruction-level power models. However, only a

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

: estimated power, HD_i: Hamming Distance for signal i,
α_i: Power coefficient for signal i, α_0: Base power coefficient

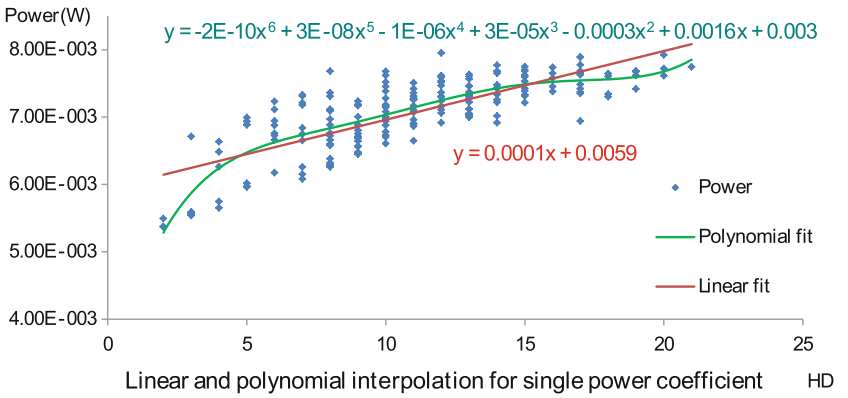


Fig. 4.21 Unit-level power model [206] Copyright ©2013 **IEEE**

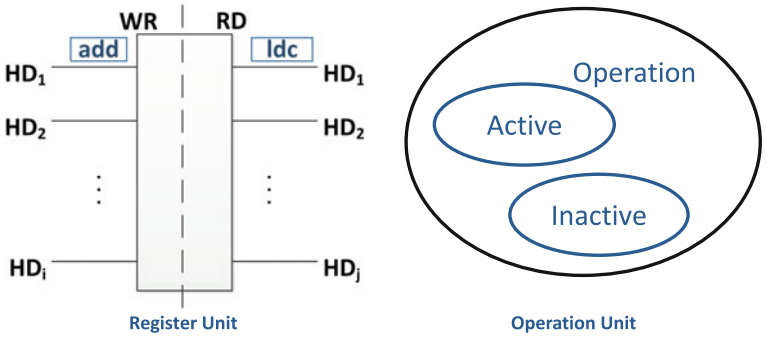


Fig. 4.22 Separate modes for power models [206] Copyright ©2013 **IEEE**

Table 4.7 Power estimation accuracy for each instruction group [206] Copyright ©2013 **IEEE**

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

few additional units (which should be usually less than the number of pipeline stages) are added into processor pipeline to support the new instruction. This greatly reduces the effort of characterization, which enhances the flexibility of proposed flow.

Static power The proposed modeling approach realizes the contribution of static power as the constant value of base power coefficient in Fig. 4.21, which does not scale with any hamming distances.

4.4.1.4 Experimental Results in Power Modeling

Instruction-level Power The accuracy of instruction-level power is demonstrated for synthesized RISC processor under 90 nm technology node at 500 MHz. The average error of power by the proposed method and that with gate-level power simulation by PrimeTime PX are present in Table 4.7. Except load instruction *ld_rr*, inaccuracy for all instruction groups are below 5%. The load instruction is relatively inaccurate (7.89%) due to the memory interface module, which is not included in the core architecture.

Instruction-level power models are straightforward to extend to other processor models. Power models for another RISC processor with mixed 16/32 bits ISA of 6 pipeline stages, synthesized under four different technologies, are further characterized. The processor is synthesized at 25 MHz. Figure 4.23 presents the instruction level power consumption.

Application-level power Six embedded applications are applied to demonstrate the speed and accuracy of proposed power modeling flow. Figure 4.24 shows the instruction profiling for applications in top half and accuracy/speed comparison in the bottom half. LISA-based power estimation shows a close match with the gate-level simulation for all applications, out of which Sieve has relatively larger mismatch due to its larger percentage of memory load instructions, which is shown as less accurate in Table 4.7. It is worth noticing that high-level power estimation in ISS has its limitation to further improve accuracy since a lot of signals existing in RTL/gate-level are trimmed out in high-level representation. Such signals, which are neglected in high-level models, still contribute to overall power consumption. With regard to

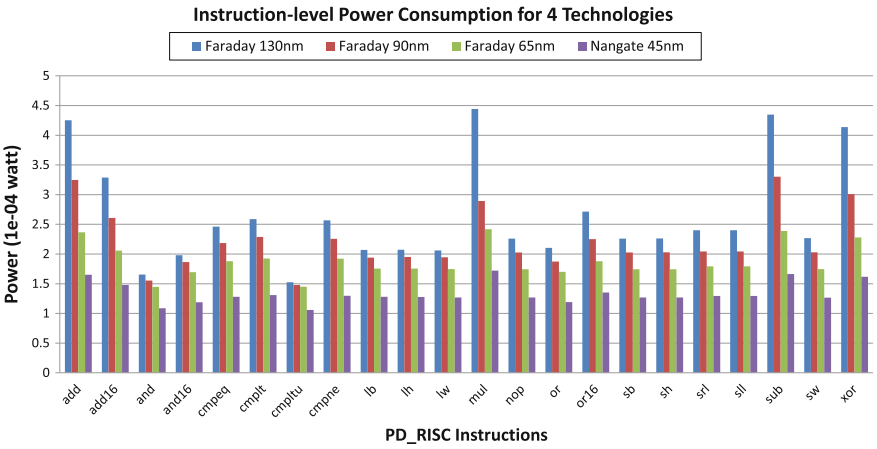


Fig. 4.23 Instruction-level power for RISC processor

estimation speed, 28x speed-up is achieved in average for all applications, which is mainly caused by the intrinsically fast speed in ISS simulation compared with low-level ones [112]. Compared with pure behavioral simulation in ISS, the simulation overhead is caused by the cycle-based activation analysis of operation units, the calculation of hamming distance and unit-level power consumption.

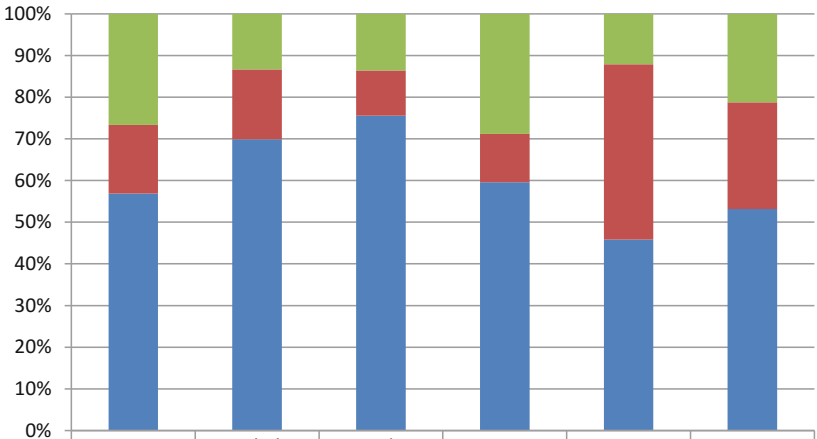
Figure 4.25 visualizes the instantaneous power consumption for both levels of four applications, which shows a dynamically close match. The resolution of proposed accurate power modeling is in one clock cycle, which is quite advanced among state-of-the-art technologies.

Power for custom instruction The Zero Overhead Loop (ZOL) instruction is implemented on the RISC processor to demonstrate the flexibility of proposed modeling approach. ZOL is used to replace explicitly specified jump instructions to create a loop of subsequent instructions with a fixed amount of iterations. In the processor model, one additional *ZOL_control* operation is created in *FETCH* pipeline stage for setting program counter and detect end of iteration. Another modification is in the *DECODE* stage where ZOL instruction needs to be included in the ISA. Power modeling for updated architecture characterizes *ZOL_control* and the updated *DECODE* operations. Table 4.8 presents a modeling error of 3.63%.

4.4.2 LISA-Based Thermal Modeling

4.4.2.1 Thermal Modeling Using HotSpot

HotSpot is an open source package for temperature estimation of architecture-level units. It has been applied in both academia and industry for architecture-level thermal



	LMS	sobel	cordic	conv3x3	sieve	FFT
Branch	26.64%	13.36%	13.57%	28.83%	12.10%	21.25%
Load/Store	16.51%	16.74%	10.85%	11.60%	42.07%	25.59%
ALU	56.85%	69.90%	75.58%	59.57%	45.83%	53.16%

App	Average Power(mW)		Error	Speed(s)		Faster
	LISA-based Simulation	Gate level Simulation		LISA-based Simulation	Gate level Simulation	
LMS	8.70	8.80	**1.40%**	1.23	40.57	**X 32.97**
sobel	9.20	9.70	**5.04%**	2.56	66.35	**X 25.87**
cordic	8.20	8.40	**2.75%**	0.79	32.59	**X 41.02**
conv3x3	8.70	8.84	**1.58%**	13.60	219.29	**X 16.13**
sieve	8.30	9.61	**13.99%**	11.40	215.81	**X 18.92**
FFT	8.80	9.30	**5.38%**	22.00	726.27	**X 33.01**
Average						**X 27.99**

Fig. 4.24 Application profiling and average power [206] Copyright ©2013 **IEEE**

modeling 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 circuit 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 modeling please refer to [86].

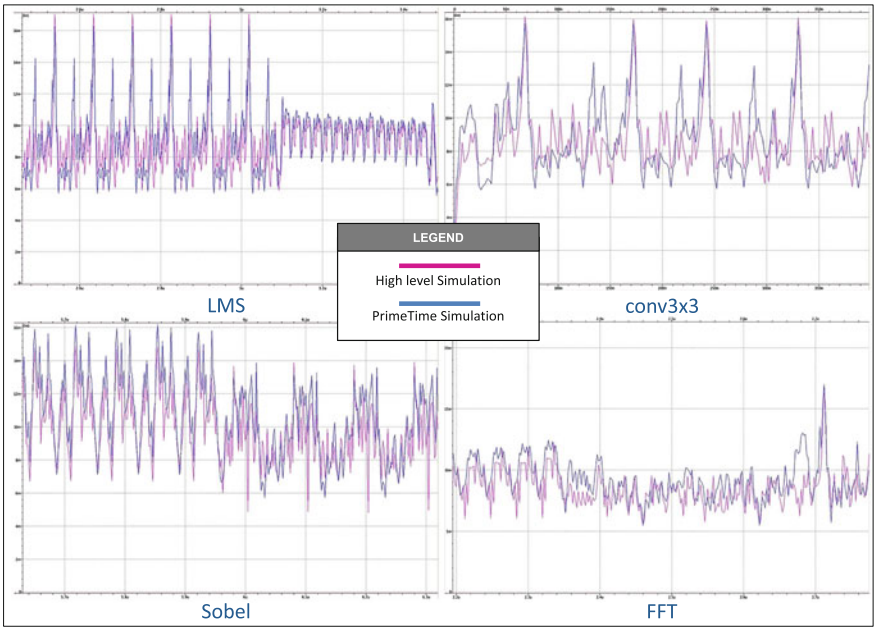


Fig. 4.25 Instantaneous power for selected applications [206] Copyright ©2013 **IEEE**

Table 4.8 Power estimation for custom instruction [206] Copyright ©2013 **IEEE**

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

4.4.2.2 Integration of Power Simulator with HotSpot

The integration of LISA power simulator with HotSpot generally follows the guideline in [83]. Two phases are required, the initialization and runtime phases, which are briefly explained in the following.

- 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. Afterward, the initial temperature is set by the user. For instance, 60 °C is initialized for starting temperature while 45 °C is set as ambient temperature. Figure 4.26 shows an example of the floorplan information of an 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 the logic synthesizer. The data in Fig. 4.26 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

```
# 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
```

Fig. 4.26 Floorplan information for input of HotSpot framework

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 ms. For different clock frequency, the same interval is maintained to make a fair comparison. The power values which provide to HotSpot are the average values of the previous sampling interval.

4.4.2.3 Temperature Simulation and Analysis

Figure 4.27 shows an example of the runtime temperature simulation for BCH application under a synthesis frequency of 500 MHz. The unit of time is in the nanosecond while the temperature is in Degree Celsius.

Table 4.9 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 Fig. 4.26 is applied for all frequencies. As the power increases dramatically with frequency, the temperature shows slightly increment for most of the units such as DC, MEM, and WB. Rapid increment lies between 100 and 500 MHz 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 of such units due to their small area provided by the floorplan.

Table 4.9 shows the temperature for BCH application using different floorplans. The first floorplan as in Fig. 4.26 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

Time	Temperature (Degree Celsius)									
(nanosec)	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

Fig. 4.27 Instantaneous temperature generated by HotSpot

Table 4.9 Temperature and power of LT_RISC at different frequencies running BCH application

Frequencies	25 MHz		100 MHz		500 MHz	
Units	Temp (°C)	Power (mW)	Temp (°C)	Power (mW)	Temp (°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

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°. In other words, it reflects a maximal power density around $2.00\,W/m^2$ for arbitrary logic units. According to the strong relationship of temperature with power density, further thermal optimization techniques could be purposed (Table 4.10).

Table 4.10 Temperature of LT_RISC running BCH application using different floorplans

Units	Power @500 MHz (mW)	Floor plan 1			Floorplan 2		
		Size (mm^2)	Power density (W/m^2)	Temp (°C)	Size (mm^2)	Power density (W/m^2)	Temp (°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 4.11 Temperature of LT_RISC at 500 MHz for different applications

Units	Temperature (°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 4.11 shows the temperature of processor units by end of the simulation time for 10 embedded benchmarks using the initial floorplan in Fig. 4.26. 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 μs 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 applica-

tion, 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 *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.

4.4.3 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) [5] and Inverted Temperature Dependence (ITD) [98]. Most of the previous work focus on device and gate-level. Such effects can be modeled using the architectural level thermal simulation framework proposed in this work so that a thermal delay simulator for generic processor architecture could be easily generated and explored.

Figure 4.28 shows the integration framework with power and thermal simulator to model the delay fault. As discussed in Sect. 4.4.2, the LISA-level temperature simulator is generated using power simulator, HotSpot package, and architectural floorplan. Thermal directed delay fault is modeled by combining the thermal simulator and the high-level timing fault injection discussed in Sect. 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 modeled according to a second order polynomial model for 65 nm technology. The effect of ITD for different applications running on an RISC processor is also presented.

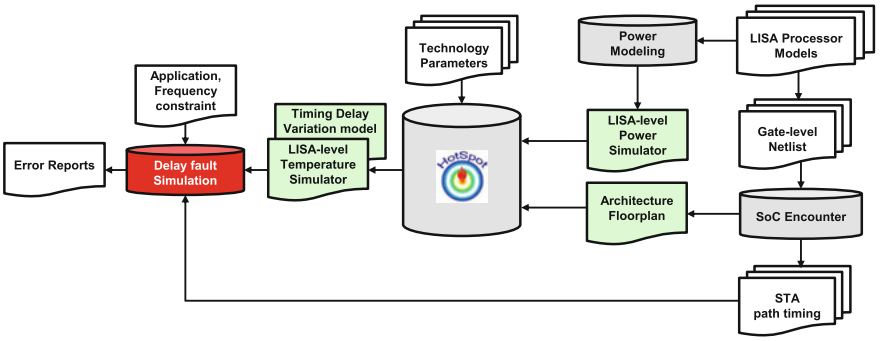


Fig. 4.28 Thermal-aware fault injection

4.4.3.1 Inverted Temperature Dependence

Propagation delay of CMOS transistor is widely modelled using the *Alpha-power* law [165] as:

$$Delay \propto \frac{C_{out} V_{dd}}{I_d} = \frac{C_{out} V_{dd}}{\mu(T)(V_{dd} - V_{th}(T))^{\alpha}} \tag{4.3}$$

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}, the delay is less sensitive to the term $V_{th}(T)$ but to the mobility, while at low temperature the thermal effects on threshold voltage dominate 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.

4.4.3.2 Timing Variation Function for Inverted Temperature Dependence

The effects of ITD for 65 nm technology are modeled using the trend of delay change for clock tree network in [167]. Two assumptions are made to simplify the high-level modeling:

1. The delay of logic path follows the same ratio of temperature/voltage dependency of the 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 modeled currently.

Figure 4.29 shows two critical paths for he 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*, *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.

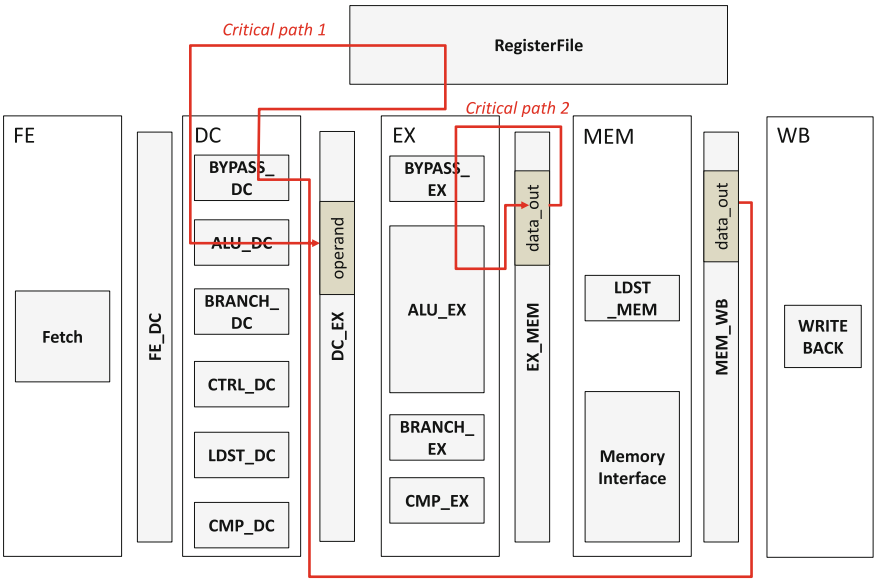


Fig. 4.29 Critical paths and transverse blocks

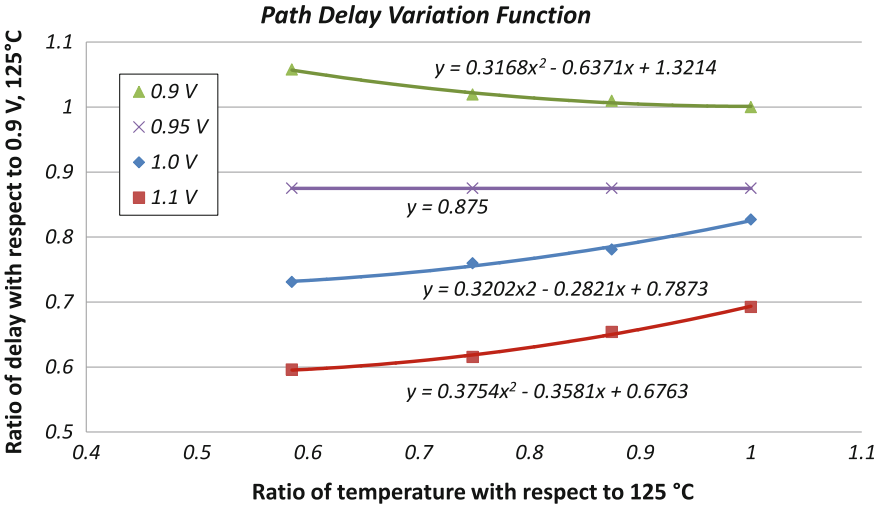


Fig. 4.30 Delay variation function under several conditions

With the above assumption and the referred data for 65 nm technology in [167], the second order polynomials shown in Fig. 4.30 are interpolated to represent the relationship between the supply voltage, instantaneous temperature, and propagation delay.

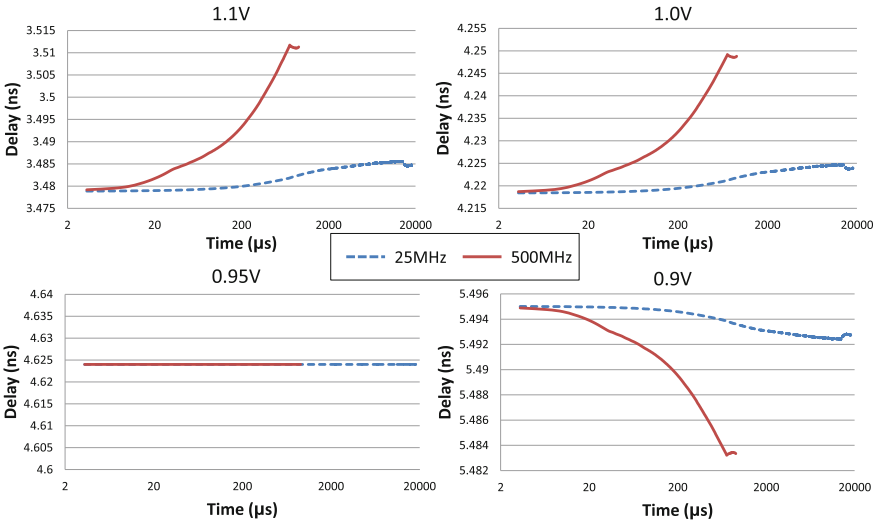


Fig. 4.31 Runtime delay of critical path for BCH application

It is observed that the trend of propagation delay with temperature differs with supply voltage. For 1.0 and 1.1 V the delay increases with temperature while decreases at 0.9 V. In [167] the ZTC voltage is known to be 0.95 V for 65 nm technology from STMicroelectronics, which proves the effect of ITD for advanced technology.

4.4.3.3 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 4.31 shows the runtime delay of the critical path for the RISC processor running BCH application. Curves are plotted for both frequencies of 25 and 500 MHz. The supply voltage is simulated using 0.9, 0.95, 1.0 and 1.1 V. The initial delay of critical path extracted out of the timing analysis tool is for the worst case condition under 125 °C, 0.9 V. It is observed that for high supply voltage such as 1.1 and 1.0 V, the delay increases with temperature till a saturation point then slightly decreases according to the characteristics of the application. For a low voltage of 0.9 V, 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 0.95 V, the delay is not affected by the temperature as expected. The effect of ITD shows the potential of frequency over-scaling under lower voltage, which is predicted for 65 nm and further technologies in [219]. With regard to different running frequencies, the processor running at 500 MHz consumes higher power which leads to higher temperature compared to the data at 25 MHz. Consequently, the speed of delay change shows more significant dependence on temperature for higher frequencies.

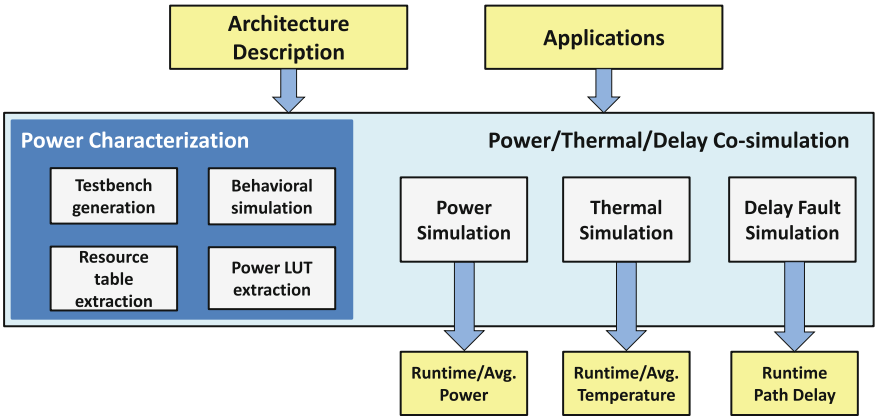


Fig. 4.32 Automation flow of power/thermal/logic delay co-simulation

4.4.4 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 [184]. Furthermore, the overheads for both characterization and simulation are discussed.

4.4.4.1 Flow Summary

Figure 4.32 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 the 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, the hash container is applied to increase the speed of instruction-architecture specific LUT addressing. The hierarchical power data according to Fig. 4.20 is dumped during simulation. More modeling architecture units lead to a higher overhead of power estimation.

Thermal and delay simulations are automatically generated once upon power simulator is ready since no further characterization steps are 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.

4.4.4.2 Overhead Analysis

Table 4.12 shows the timing and accuracy for power characterization phase under two groups of testbenches, where 10 architecture units are modeled. The first group consists of 14 types of instructions to cover the most generalized processor instructions. For instance, ALU instructions such as add and sub 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 4.12, 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 modeling of embedded processors.

Table 4.13 represents the runtime overhead of different simulation mode including pure behavioral simulation, power estimation, thermal estimation and delay simulation, where 10 architecture units are modeled. It is observed that the runtime overhead significantly lies in the power estimation compared with behavioral simulation, on which details have been discussed in Sect. 4.4.1.4. The thermal simulator achieves

Table 4.12 Time and accuracy of power characterization for testbench groups

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

Table 4.13 Runtime overhead for different simulation modes

Simulator applications	Behavior (s)	Power (s)	Times	Thermal (s)	+%	Delay (s)	+%
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

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 the 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.

4.4.5 Summary

In this work, a processor power/thermal/delay joint modeling 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 high-level reliability estimation techniques are illustrated which fast characterize the effects of errors on processor architecture. In Sect. 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 Sect. 5.2 Probabilistic Error Masking Matrice is introduced to predict error effects through the graph network of dynamic processor behavior. In Sect. 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 the behavior of circuits under faults. Mukherjee et al. [138] introduced the concept of architecturally correct execution (ACE) to compute the vulnerability factors of faulty structures. In [21] the authors performed the ACE analysis to compute architectural vulnerability factors for cache and buffers. Recently, Rehman et. al [153, 164] extended the ACE concepts to instruction vulnerability analysis and proposed reliability-aware software transformations. The vulnerability of the instruction is analyzed in this work by studying the constituent logic blocks and possibly connect with the circuit-level reliability analysis [162]. While the instruction vulnerability index model proposed at [153] includes the logical masking effects, the details of the derivation of the masking effect are not mentioned. The simulation accuracy is compared with other software-level reliability estimation flows [153].

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_5

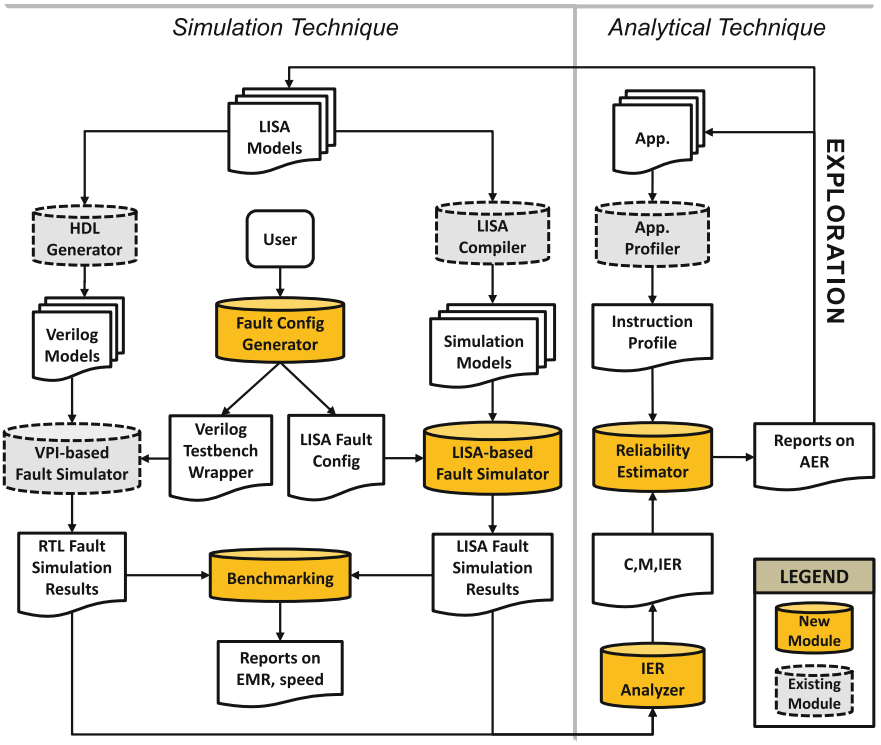


Fig. 5.1 ADL driven reliability estimation flow [216] Copyright ©2013 **IEEE**

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 an instruction set simulation-based fault simulation technique in Sect. 4.1. Figure 5.1 shows the contributions where the novel modules are filled in the 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.

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 [184].

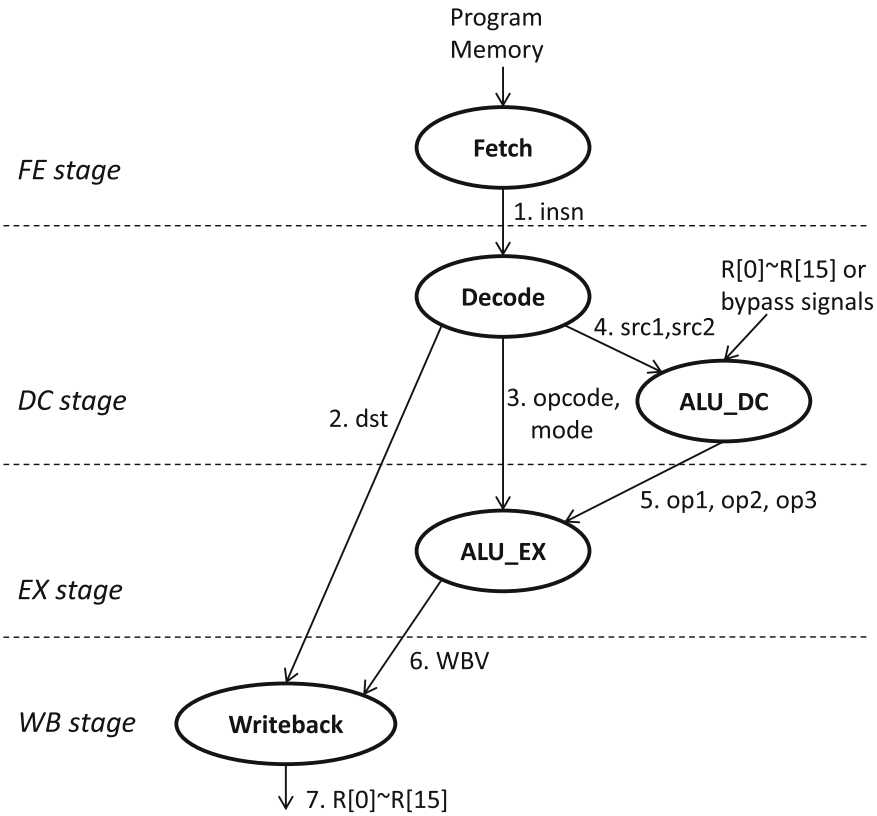


Fig. 5.2 Data flow graph for ALU instruction [216] Copyright ©2013 **IEEE**

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.

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 a 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 Eq. 5.1 shows the instruction error rate when operation Fetch in Fig. 5.2 is fault injected. The edges in the equation are labelled by their indexes.

$$\begin{aligned} IER_{alu}^{fetch} = & C_{fetch}^{1} M_{decode}^{1,2} M_{writeback}^{2,7} + \\ & C_{fetch}^{1} M_{decode}^{1,3} M_{alu\_ex}^{3,6} M_{writeback}^{6,7} + \\ & C_{fetch}^{1} M_{decode}^{1,4} M_{alu\_dc}^{4,5} M_{alu\_ex}^{5,6} M_{writeback}^{6,7} \end{aligned} \tag{5.1}$$

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 result 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 a 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.

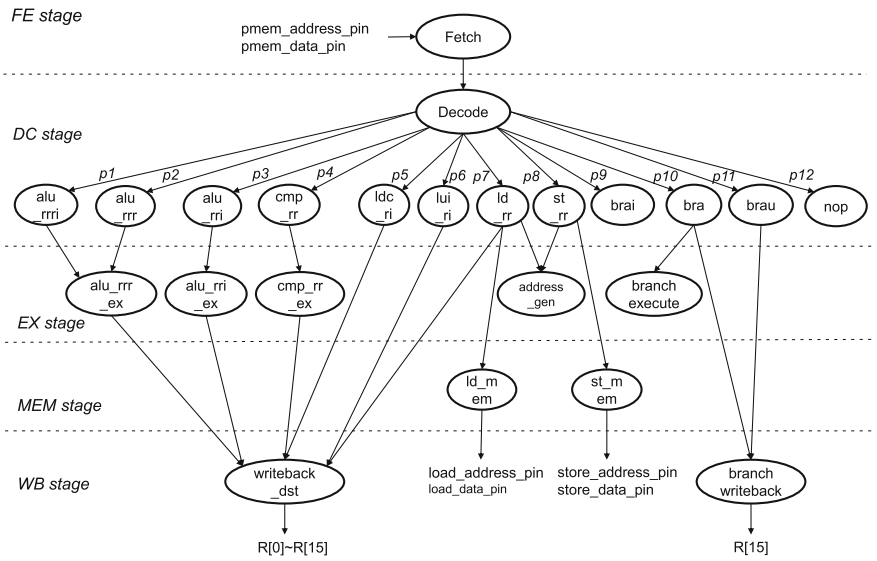


Fig. 5.3 Operation graph for all instructions in RISC processor [216] Copyright ©2013 **IEEE**

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 labeled 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 Eq. 5.2. The summation happens since the operation is on the activation chain of two instructions *alu_rrr* and *alu_rrri*.

$$AER_{app}^{alu\_rrr\_ex} = p_{app}^{1} IER_{alu\_rrri}^{alu\_rrr\_ex} + p_{app}^{2} IER_{alu\_rrr}^{alu\_rrr\_ex} \tag{5.2}$$

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 instructions. 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.

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. The 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 Eq. 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 Eq. 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.

5.1.4.2 AER

When $IER_{insn}^{op_faulty}$s for all operations and instructions are obtained from the testbenches, Eq. 5.2 is applied to estimate $AER_{op\_faulty}^{app}$ based on the application profil-

Table 5.1 Instruction-level reliability estimation [216] Copyright ©2013 **IEEE**

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

ing. 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 a 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 h* for IDCT. This is a significant improvement in the productivity and facilitates exploration by the application developer like the optimizations proposed in [164]. 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_rri_dc* and *alu_rri_ex* since the compiler generates more assembly codes for calculation with immediate values.

Table 5.2 Reliability estimation for selected applications [216] Copyright ©2013 **IEEE**

Operation	$AER^{app}_{op\_faulty}$						Rel. Dev.
	Sobel		FFT		IDCT		
	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

For estimation accuracy, the results of 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 is demonstrated through several embedded applications on an RISC processor and by benchmarking against an high-level fault injection.

5.2 Probabilistic Error Masking Matrix

The design of reliable system in presence of faults is a challenging problem, which requires the understanding of the causes and effects of failures such as radiation and electromigration. Moreover, reliability trades off with other design metrics [2, 59, 91, 160]. Recent research shows that separate error mitigation techniques from individual design abstractions may lead to over-protected system. Therefore it is desirable to treat reliability as a *cross-layer design issue* [47]. For instance, the architectural tolerant fault technique should take advantage of circuit-level and algorithmic error resilience [74, 146]. However, cross-layer exploration requires clear knowledge of the fault propagation through design abstractions. Using such knowledge, error properties such as injection time, location and probabilities could be approximately predicted even before tedious fault injection experiments.

In particular approximate error prediction is important for *algorithmic reliability* and *inexact, probabilistic computing* [141]. Earlier research on this can be traced to the issue of floating-to-fix point conversion for DSP design [75]. However, there the error locations are limited to variables (sizes of fixed points) and operators (saturation, rounding effects), which neglects the concern on architectures. The framework of Probabilistic Transfer Matrix (PTM) proposed by Krishnaswamy [162] captures the probabilistic behavior of the circuit to estimate the error probability **inside** the circuit. However, PTM suffers from scalability issue for large design due to its granularity of single bit. In [131] a statistical error tracking approach named RAVEN is introduced to analyze cross-layer error effects. The DUE (Detected Unrecoverable Error) and SDC (Silent Data Corruption) outcomes for soft errors are predicted by RAVEN. 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 novel algebraic representation called Probabilistic error Masking Matrix (PeMM) is proposed to address the masking effects on errors occurring at the **inputs** of the circuits. In contrast to the high computational complexity of PTM, PeMM requires very few calculation since it has initially granularity on the signal level. Fine-grained PeMM is also designed to calculate nibble-wise or byte-wise error probabilities. In the next, PeMM algebra has been integrated into LISA-based high-level processor design framework, where logic errors are represented as an abstract data structure of *token*. An automated analysis flow predicts the token propagation by a cycle-accurate instruction set simulator while PeMM

addresses the error masking effects for micro-architecture components. Several optimization techniques are introduced to increase the prediction accuracy, which heavily depends on the control states of the architecture.

5.2.1 Logic Masking in Digital Circuits

Faults within logic circuits are masked with certain probability before propagating as output errors. Such masking effects are caused by:

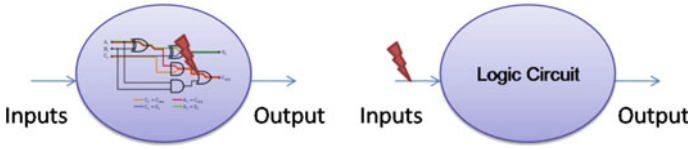
- Logic primitives containing arithmetic operators have inherent error masking abilities.
- Micro-architecture features can ignore the erroneous data, such as data bypassing and branch prediction.
- Errors in architecture resources such as registers and memory elements can be never used or overwritten before being read

PTM [162] calculates error probability of outputs for faults inside the circuits Fig. 5.4a. It suffers from scalability problem since PTM has the matrix size of $2^n \times 2^m$ where n and m are the total number of bits for inputs and outputs. Derivation of PTM for large design is performed by accumulating PTMs for individual logic gates, which is infeasible for modern VLSI.

Probabilistic error Masking Matrix (PeMM) primarily handles the case in Fig. 5.4b where the faults locate at *inputs* of circuits. PeMM only has the matrix size of $m \times n$ for a circuit with n bits input and m bits output. PeMM can be further compressed when n and m represent the number of input and output signals.

5.2.1.1 PeMM Definition

For a circuit with n inputs and m outputs which are labelled as $in_0, \ldots in_{n-1}$ and $out_0, \ldots out_{m-1}$ respectively, the PeMM P of the circuit has a dimension of $m \times n$. Each element $p(out_i, in_j)$ indicates the error probability on output out_i with regard to input in_j with 100% error, where $i \in [0, m-1]$ and $j \in [0, n-1]$. $p(out_i, in_j)$ equals 0 represents a complete error masking while 1 implies no masking at all.



(a) Faults inside circuit (b) Faulty inputs

Fig. 5.4 Faults in logic circuits [207] Copyright ©2015 **IEEE**

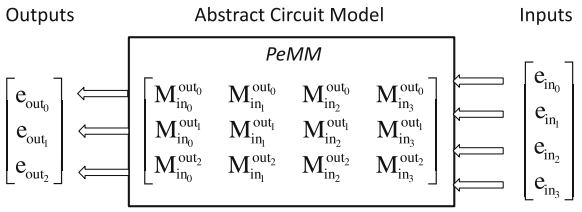


Fig. 5.5 Probabilistic error Masking Matrix (PeMM) [207] Copyright ©2015 **IEEE**

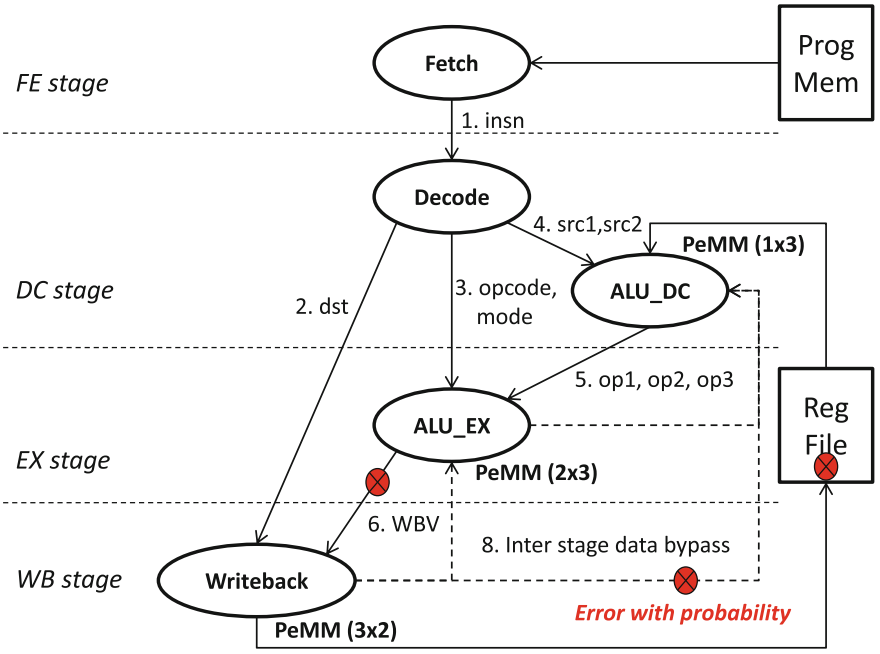


Fig. 5.6 Logic blocks involved for ALU instruction [207] Copyright ©2015 **IEEE**

$e_{out_i} \in [0, 1]$ implies the truncation of error probability when it is larger than one. Elements in inputs $I(j)$ represents the error probability e_{in_j} on input in_j. The output vector is vector with dimension $m \times 1$ with elements showing the error probability e_{out_i} on the output. Figure 5.5 visualizes the PeMM for abstract circuit model.

PeMMs characterize error masking effects of micro-architecture components. The circuit PeMM is evaluated as the concatenation of PeMMs for sub-components. The architecture components for the ALU instructions and data signals among them are shown in Fig. 5.6. The dimensions of component-wise PeMMs are labeled in bold color. The propagated tokens are indicated by the rounded red dot, which represents error data with probabilities.

5.2.2 PeMM for Processor Building Blocks

5.2.2.1 Combinational Logic Blocks

PeMM tackles the masking effect of the circuit by a linear transformation. However, such approach does not handle the logic blocks with internal data dependencies. One solution is to decompose larger circuits into logic sub-blocks with individual PeMMs according to their data dependencies. Figure 5.7 indicates PeMM decomposition of large logic block *alu_ex* into 3 sub-blocks. Signals *alu_in1* and *alu_in2* connect *alu_ex_1* and *alu_ex_2* while *alu_out* connects *alu_ex_2* and *alu_ex_3*. Following this approach, PeMMs for sub-blocks with no data dependencies inside can be characterized individually. An intra-token pool is used to keep the temporary tokens for further processing inside large logic blocks. The intra pool shows the fact that such tokens can not be accesses by other logic blocks.

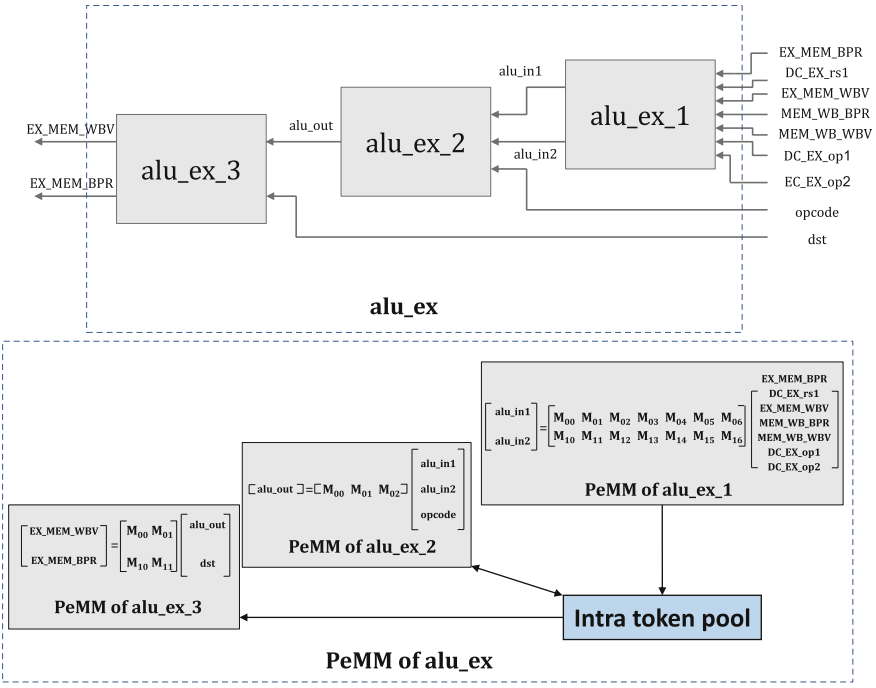


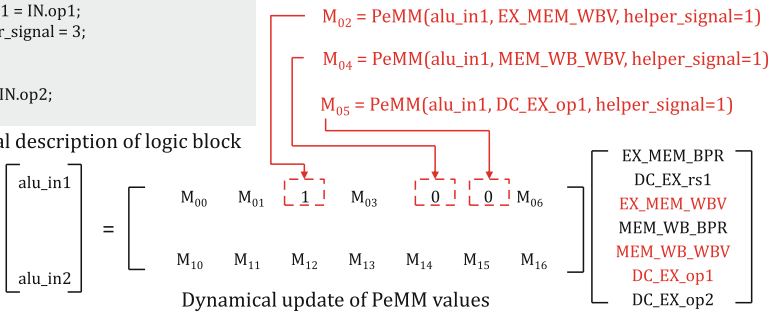
Fig. 5.7 Decomposition of large logic block using PeMM [207] Copyright ©2015 **IEEE**

```
BEHAVIOR
{
  if (BYPASS_ACTIVE(MEM.IN.BPR, IN. rs1))
  {
    alu_in1 = MEM.IN.WBV;
    helper_signal = 1;
  }
  else
  {
    if (BYPASS_ACTIVE(WB.IN.BPR. IN.rs1))
    {
      alu_in1 = WB.IN.WBV;
      helper_signal = 2;
    }
    else
    {
      alu_in1 = IN.op1;
      helper_signal = 3;
    }
  }
  alu_in2 = IN.op2;
}
```

Behavioral description of logic block

helper_signal	MEM.IN.WBV	WB.IN.WBV	IN.op1
1	1	0	0
2	0	1	0
3	0	0	1

PeMM LUT w.r.t. value of helper signal



Dynamical update of PeMM values

Fig. 5.8 Control flow handling for PeMM [207] Copyright ©2015 **IEEE**

5.2.2.2 Control Flow Inside Logic Block

Non-linear operators inside logic block such as multiplexers generated from control flow reduce the prediction accuracy of PeMM. This results in a significant difference of masking probability compared with random characterization. For instance, the circuits shown in Fig. 5.8 contains a 3-to-1 multiplexer from the conditional statements. Random characterization for the highlighted PeMM elements give the value of [0.33 0.33 0.33], which is false for the real masking due to the exclusiveness of multiplexer. To solve this, additional *helper_signals* are declared to indicate dynamically active branch and fill the correct PeMM elements. For example, vector [1 0 0] is filled when the first branch of the if statement is active, which correctly shows that the error from the first branch propagates to output directly and errors on other branches are masked completely.

5.2.2.3 Sequential Logic and Memory

Sequential logic (RegisterFile and pipeline registers) and memory block show no logic masking effects on their inputs. Identity Matrix $I_{m\times m}$ can be used to model PeMM directly, where m is the number of inputs and outputs. For pipeline registers, errors on inputs are propagated to outputs during pipeline shift. For RegisterFile, input

errors are stored during write access and loaded during read. Similarly, PeMM for memory is modeled by identity matrix with m equalling to the count of storage cells. Noted that sequential logic has strong timing masking effect, such as timing error caused by setup/hold violation. Such factor is currently not containing in behavioral PeMM and will be integrated during future work.

5.2.2.4 Inputs with Multiple Faults

Multiple errors on PeMM inputs also affects its accuracy. Matrix multiplication with input vector accumulates the contribution of all input errors, which achieves good masking accuracy for most arithmetic operators. However, correlated input errors which are partially or completely generated from the same error can cancel their error effects depending on the arithmetic operators. For instance, a strong error cancellation effect exists for XOR operator with bit-flip errors at same bit position of both inputs. Ideally, for multiple input errors, a new set of PeMM should be adopted which gives additional modeling effort. However, since such case is relatively rare, PeMMs with single input errors are still applied to give a worst case estimation.

5.2.3 PeMM Characterization

Statistical simulation is used to characterize PeMM elements when primary inputs of logic blocks are injected with errors. High-level languages such as C based test-benches embeds behavioral description of circuit. The probability $M_{in_j}^{out_i}$ can be acquired by averaging the error probability on out_i among multiple experiments, where randomly single bit-flip error is injected on input in_j.

5.2.3.1 Accuracy of PeMM Characterization

In order to characterize the PeMM elements with the desired confidence level, the number of random experiments is determined according to [42] by randomizing input values and bit position of errors. For a circuit under test with n inputs of m bits each, the space size of input randomness is $2^{m \times n}$. The overall size of random experiments with random bit error position equals $2^{m \times n} \times m$. For example, a circuit with 2 inputs of 32 bits for each, needs 9604 experiments to produce the PeMM element for 95% confidence level with confidence interval of 1.

5.2.3.2 Fine-Grained PeMM

To trade off prediction accuracy and modeling complexity, PeMM can be extended to model errors on finer granularities, such as byte or nibble levels. Therefore, not

Table 5.3 Examples of PeMM elements with byte-level granularity [207] Copyright ©2015 **IEEE**

Operation	Key	Byte-wise $M_{in_j}^{out_i}$			
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

only existence of errors on signal can be predicted but also the error distribution across the bits of signal. This can be of importance for prototyping of algorithms and architectures for approximate computing.

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 1^{st} byte of first input and 3^{rd} byte of second input while key 10 shows no fault in second input but only 1^{st} 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 1^{st} byte of *SUB* operation can result in errors in 2^{nd} or even 3^{rd} bytes with reduced probability, whereas for AND operation no cross bytes error could result 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.

Figures 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 overall error probabilities on a specific segment of signal *out* are 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 neighbor segments with reduced error probabilities once upon fault is injected in a single segment. Furthermore, nibble-level PeMM shows the cross-section error propagation more clearly since mismatches on finer segments are characterized.

```
|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 ------------------------------------------------------------------------------------- |
|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    |
|--------------------------------------------------------------------------------------------|
```

Fig. 5.9 Byte-level PeMM

```
Level_2 ----------------------------------
alu_out/opcode
     0.869     0.898         0         0         0         0         0         0
     0.898     0.894         0         0         0         0         0         0
     0.899     0.899         0         0         0         0         0         0
      0.88     0.904         0         0         0         0         0         0
     0.885     0.893         0         0         0         0         0         0
     0.893     0.908         0         0         0         0         0         0
     0.904     0.895         0         0         0         0         0         0
      0.88     0.867         0         0         0         0         0         0


alu_out/alu_in1
      0.79         0         0         0         0         0         0         0
     0.088     0.799         0         0         0         0         0         0
     0.007     0.094     0.788         0         0         0         0         0
         0     0.006     0.089     0.796         0         0         0         0
         0         0     0.003     0.087     0.801         0         0         0
         0         0         0     0.004     0.104     0.817         0         0
         0         0         0     0.001     0.012     0.079     0.783         0
         0         0         0         0     0.002     0.003     0.087      0.79


alu_out/alu_in2
     0.799         0         0         0         0         0         0         0
     0.106     0.811         0         0         0         0         0         0
     0.008     0.088     0.776         0         0         0         0         0
     0.002     0.005     0.105     0.807         0         0         0         0
         0         0     0.007     0.099     0.795         0         0         0
         0         0         0     0.009     0.088     0.803         0         0
         0         0         0         0     0.006     0.092      0.81         0
         0         0         0         0         0     0.009     0.088     0.792
```

Fig. 5.10 Nibble-level PeMM

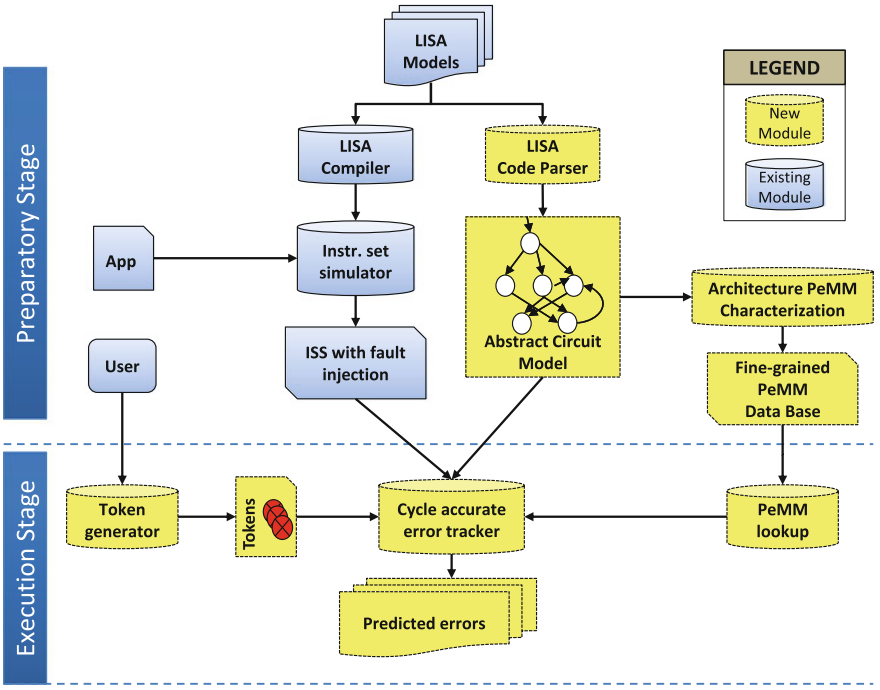


Fig. 5.11 Error tracking and prediction framework [207] Copyright ©2015 **IEEE**

5.2.4 Approximate Error Prediction Framework

The PeMM based algebraic operation is integrated with LISA-based processor design flow [184] to establish an approximate error prediction framework for generic architecture. Other simulators using ADL such as Verilog and SystemC can also take advantage of this technique. Figure 5.11 presents an overview of the framework.

The flow is composed of the preparatory and execution stage. In preparatory stage, cycle accurate instruction-set simulator (ISS) is generated from processor description using ADL LISA [1] with user provided applications. The simulator is extended with fault injection technique as in Sect. 4.1. An additional parser of LISA source code is used to extract the behavior section of LISA operations and the inputs and outputs resources for individual architecture units. The PeMM characterization module wraps the behavior of processor architecture components into C-based testbenches with interface signals as function arguments. PeMMs are fast characterized in such testbenches with random inputs and faults. The LISA parser supports language pragmas for extended PeMM characterization according to Sects. 5.2.2.1, 5.2.2.2 and 5.2.3.2.

In the execution stage, the user injects token with graphical interface or with description by XML file. The token data structure indicates error probability, along

with elements representing the micro-architectural location and timing which are required to track the token during propagation. PeMMs algebra is called by active logic units to calculate output error probabilities, while inactive logic units completely mask their input token. The final report contains predicted errors by the end of simulation, as well as the detailed paths of token propagation and error masking conditions.

5.2.4.1 Error Representation

Compared to faults and errors, token injection does not alter the resource values but annotate an error probability which is initially set to one. The token is removed when its error probability is masked to 0. To fetch the correct token, hardware resource ID and array index are updated together with error probabilities. Specific hardware resources are able to contain multiple *sub-tokens*. For instance the *instruction* register contains sub-tokens in each of its decoding fields such as *opcode*, *source* and *destination* operands.

5.2.4.2 Token Tracking

As no actual errors are injected by the tokens, the simulator remains correct execution and indicates potential errors. Algorithm 1 describes the token tracker called at each clock cycles. The algorithm begins with the activation checking of LISA operations. If any activated operation has inputs containing tokens, PeMMs are applied to update and propagate tokens to the outputs. Due to synchronized hardware behaviors, the tokens are scheduled for creation and removal at the end of that cycle. Besides activation analysis for operations, the tokens in pipeline registers are forwarded to the next pipeline stage. However, forwarded tokens are overwritten by the ones created from the active operations if there is any. Old tokens in memory and register files are replaced by new ones if they are not read out before overwritten.

5.2.5 Results in Error Prediction

Several case studies on an embedded RISC processor from Synopsys Processor Designer [184] are used to demonstrate the proposed approximate error prediction framework. The processor has five pipeline stages with full data bypassing and forwarding functionality. Both RTL models and simulators are generated automatically.

Table 5.4 Example of Error Prediction Report

Token ID	Resource Name	Array Index	Created Cycle	Word Error (%)	Nibble-level error probability (%) (LS nibble-MS nibble)						
Token in bit 6 for *insn* register of *FE/DC* pipe stage @ cycle 3											
					⇓						
11	R	3	8	100	0	100	23.5	1.48	0.11	0.005	0.001	0
14	R	4	9	98.42	0	84.4	32.5	2.04	0.13	0.01	0	0
18	R	6	11	96.49	0	84.4	32.5	2.04	0.13	0.01	0	0
21	R	7	12	95.54	0	84.4	32.5	2.04	0.13	0.01	0	0
28	R	11	14	50.87	0	50.2	0	0	0	0	0	0
30	R	5	15	50.87	0	50.2	0	0	0	0	0	0
32	R	12	16	56.54	0	68.7	0	0	0	0	0	0
33	dmem	0x7FFF	17	54.27	0	49.6	50.4	0	0	0	0	0
Token in bit 1 for *WBV* register of *MEM/WB* pipe stage @ cycle 17											
					⇓						
3	dmem	0x7FFF	17	100	49.8	50.2	0	0	0	0	0	0
Token in bit 15 for *WBV* register of *EX/MEM* pipe stage @ cycle 9											
					⇓						
5	R	6	11	100	0	0	0	100	23.1	1.5	0.10	0.01
6	R	7	12	98.42	0	0	0	84.3	32.4	2.1	0.14	0.01

5.2.5.1 Error Prediction Report

Token tracking analysis is carried out on an assembly program consisting of algorithmic and memory access instructions. Tokens are created at different hardware resources where the error prediction reports are documented in Table 5.4. For instance, the first group shows that the created token with ID 1 expands into totally 33 tokens dynamically. Only 8 tokens live until the end of simulation while the rest ones have been removed or overwritten. The token in processor core is stored into data memory for memory access instruction. On the contrary, the token in second group results in one error in data memory although just 3 tokens are expanded. In the last group, although 6 tokens are expanded, no token has been stored into data memory so that no application-level errors are visible. Based on the error prediction reports the user can easily perform vulnerability analysis for specific hardware resources in the architecture.

The report also indicates word-level and nibble-level error probabilities. The two sets of error probabilities differ from each other for absolute values since they are calculated using separate word and nibble-level PeMMs respectively. It is noted that the errors are expanded into adjacent nibbles with reduced error probabilities due to the inter-nibble masking effects of algorithmic operations.

Algorithm 1 Token tracking routine [207] Copyright ©2015 **IEEE**

```
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 ∃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 ∃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
```

5.2.5.2 Accuracy and Speed-Up

The predicted error probability is benchmarked with Verilog-based fault injection [44]. The faults are able to be injected into physical resources such as RTL signals, pipeline registers, register file and memory arrays in Verilog description.

Accuracy Comparison for Different PeMM Modes In this experiment, a testbench processes data in a loop using general purpose registers and stores the final result into memory. Error prediction results with different modes of PeMM construction are benchmarked with Verilog fault injection. For each fault injection experiment, random inputs are generated with single bit-flip error at random bit position. 1,000 experiments are performed to calculate the average error probabilities on selected hardware resources. In contrast, the proposed PeMM based analysis is performed in one run to generate the predicted error probabilities under the same input error.

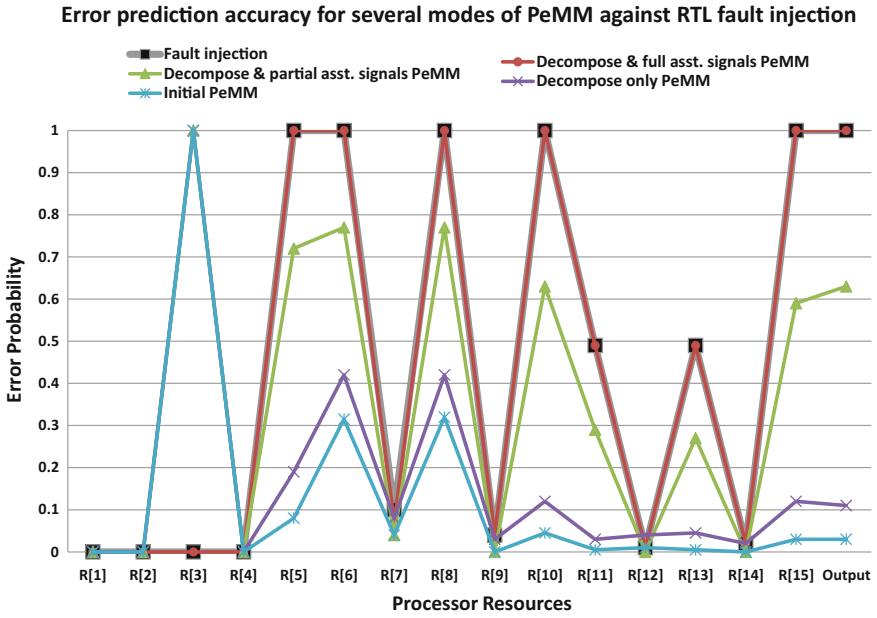


Fig. 5.12 Error prediction accuracy on different modes of PeMM against Verilog-based fault injection [207] Copyright ©2015 **IEEE**

Figure 5.12 indicates the results of prediction with different PeMM modes on selected hardware resources, which include registers R[1] to R[15] and final program output value to data memory. It is shown that the PeMM without matrix decomposition achieves the least accuracy compared with fault injection, whereas PeMM decomposition and usage of assistant signals for dynamic control flow prediction increase the accuracy significantly. Assistant signals, which cover all related logic blocks, help the PeMM to perfectly match the results of fault injection.

Error Prediction for Embedded Applications Accuracy and timing advantage of the proposed framework are demonstrated against fault injection by several embedded benchmarks. One token/bit-flip error is created/injected in the same resource location at same time instances. Table 5.5 indicates the word level error probability on selected hardware resources at then end of application. The PeMM modes are configured to be under both matrix decomposition and assistant signals.

The error probabilities of fault injection approach the analytically predicted values as a number of experiments grows. The large sample of trials during PeMM characterization phase contributes to the prediction accuracy. Table 5.5 also compares the required time between PeMM prediction and fault injection. Token tracking achieves 25,000x speed-up on average compared to fault injection of 2,000 experiments.

Table 5.5 Accuracy and speed of prediction for embedded benchmarks [207] Copyright ©2015 **IEEE**

Apps	Traced resource	Array index	Token tracker		Verilog fault injection [44]			
			Error Prob	Time (s) one token	Error Prob			Time (h)
					Number of experiments			
					1000	2000	3000	2000
CRC	dmem	0×100017	0.03	3.4	0.07	0.04	0.03	23.43
IDCT	dmem	0×100034	0.98	2.9	1.00	1.00	0.99	19.41
Viterbi	dmem	0×100078	0.90	4.4	0.98	0.95	0.93	33.30
Rijndael	dmem	0×10002F	0.48	3.9	0.61	0.66	0.59	28.24
Cordic	R	12	0.75	2.5	0.89	0.81	0.79	17.31
Sobel	R	7	0.24	2.4	0.30	0.28	0.29	16.63

Table 5.6 Processing time for automated PeMM preparation [207] Copyright ©2015 **IEEE**

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

5.2.5.3 Timing Overhead for Token Tracking

Overhead of Preparatory Stage The preparatory stage, which consists of parsing and characterization phases, generates PeMM for 42 operations automatically for the targeted processor. Table 5.6 presents the timing for the preparatory stage on the host machine of Intel Core i7 CPU at 2.8 GHz. 100,000 characterizations are performed for each element in PeMM. Characterization phase consumes larger computational efforts than parsing due to its huge amount of random experiments. Analysis of advanced PeMM modes consumes extra time in both phases.

Timing Overhead Against Number of Tokens Table 5.7 indicates the timing overhead by the token tracking against original instruction set simulation. Token tracker with no token injected adds 28.4% overhead in average due to the searching for token per clock cycle. Single injected token further increases 6.7% simulation overhead. 20 tokens add 79.3% overhead in average. During simulation, most of the tokens only have a life span of several cycles. Therefore the overhead does not scale linearly with a number of tokens. The tokens are managed in an unordered hash map with timing complexity of *O(1)*, which accelerates the searching [40].

Timing Overhead for Different Modes of PeMM Timing overhead using different modes of PeMM is present in Fig. 5.13, which indicates that run-time efforts of enhanced analysis also consumes larger overhead for all benchmarks.

Table 5.7 Timing overhead analysis against architecture simulator [207] Copyright ©2015 **IEEE**

Apps	Original Simulator [184]	Token tracker					
		0 token		1 token		20 tokens	
	No fault (s)	(s)	+%	(s)	+%	(s)	+%
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

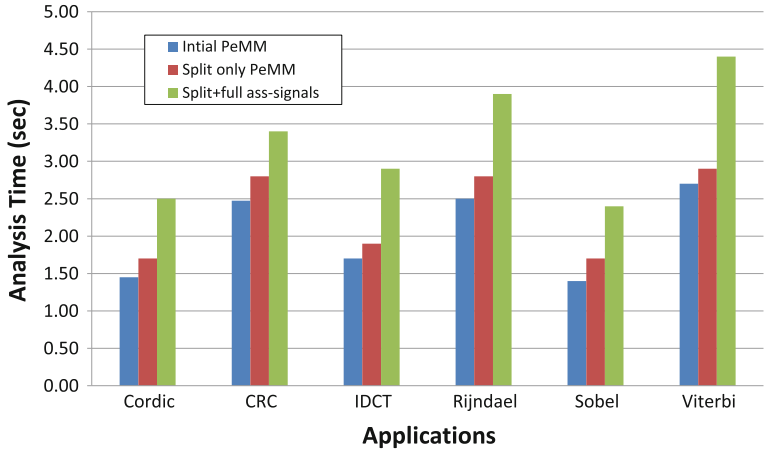


Fig. 5.13 Run-time among different PeMM modes [207] Copyright ©2015 **IEEE**

5.2.5.4 Application-level Error Locations

Another advantage of error prediction is its ability to predict error locations in the huge memory space, which is difficult to perform by fault injection. Such feature helps the designer to predict how architecture errors affect application results. The median filter [85] is demonstrated to show application-level usage of PeMM flow. Figure 5.14 shows both input and output images. Two tokens are injected in the memory locations storing selected pixels of the input image, while the accordingly effective regions are predicted in the output image. The prediction matches the algorithmic specification, where the value of each pixel in the output image has the average value of the pixels at the same position and surrounding 8 ones in the input image.

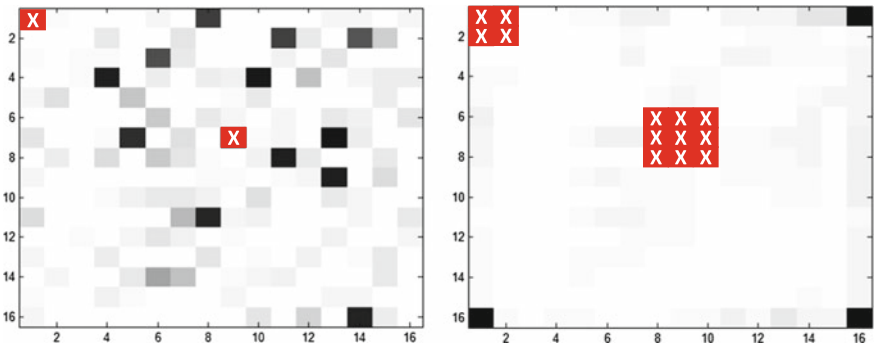


Fig. 5.14 Error prediction for median filter application [207] Copyright ©2015 **IEEE**

5.2.6 Summary

In this work, probabilistic error masking matrix (PeMM) is proposed to analyze the error masking effects of logic circuits. Integrated with PeMM algebra, an approximate error prediction framework is developed to track the path of error propagation and error probabilities. The proposed framework achieves high prediction accuracy and significant speed-up compared with state-of-the-art RTL fault injection technique.

5.3 Reliability Estimation Using Design Diversity

Redundancy is a key feature among fault tolerance techniques [102], which improves the *data integrity* of the system. Mathematically speaking, data integrity shows the probability of a system either producing the correct result or detectable errors. Hardware redundancy executes logic operations repeatedly on several hardware copies to verify the correctness. Selected works on such modular redundancy are Redundant Multi-Threading (RMT) [137, 157]. In parallel, software-based redundancy re-executes instructions when idle instruction slots are available [155].

Redundancy is constructed based on *duplication*, where two or more modules perform the same operation and evaluate the result through comparison. One metric to evaluate redundancy system is its ability against common-mode failures (CMFs), where different copies in the system are subjected to the same type of error [118].

Although fault injection [90] and analytical techniques [21] can be applied to estimate reliability, such approaches do not *quantify* the effects of CMFs on the redundant system. To address this, *design diversity* has been proposed in [12] to protect circuit-level design from CMF. In [106, 156] design diversity assists the development of robust systems. Mitra et al. [133] formally adopt it as quantifiable

evaluation metric on duplicated system. Previous works mainly apply design diversity on circuit-level designs.

Contribution This work extends the usage of design diversity from circuit-level to architecture-level through a novel graph-based analysis flow on operation exclusiveness. The proposed approach is used to quantify design diversity of various classes of architectures. The reliability of applications running on different architectures is quantified through system Mean-Time-to-Failure, which is closely related to design diversity.

5.3.1 Design Diversity

A duplex system is shown in Fig. 5.15 which consists of modules performing the same functionality. Both outputs are verified by a comparator to detect errors. Design diversity refers to the fact that different module implementations can possibly produce different outputs which are detectable facing CMFs. Figure 5.15 also shows the multiplex system consisting of more than two modules.

Assuming a pair of faults (f_i, f_j) is injected into the modules respectively. Design diversity $d_{i,j}$ of fault pair (f_i, f_j) is mathematically defined in Eq. 5.3. Provided n as the number of total input bits, 2^n is the number of all input combinations. $k_{i,j}$ is the *joint detectability*, which is the number of input combinations producing undetectable errors.

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

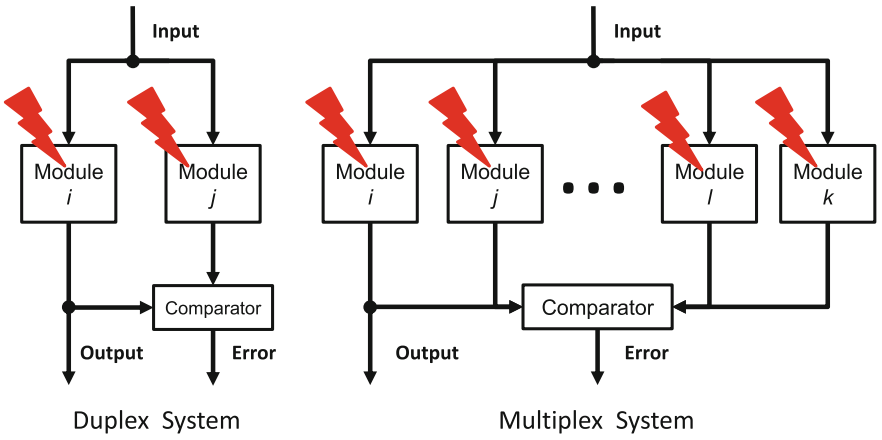


Fig. 5.15 Duplex and multiplex redundant systems [208] Copyright ©2015 **IEEE**

The design diversity of the system is defined as Eq. 5.4, which is the expected value of design diversity of all possible fault pairs. $d_{i,j}$ is the design diversity of fault pair (f_i, f_j) while $p(f_i, f_j)$ is the probability of fault pair (f_i, f_j). As system design diversity represents the probability of the system with error free or detectable errors, Eq. 5.5 shows the system error probability by simply subtracting design diversity from one.

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

$$E = 1 - D \tag{5.5}$$

Similarly, for a multiplex system design diversity $d_{i,j,..,k}$ of the fault set $(f_i, f_j, \ldots, f_k)$ is shown as Eq. 5.6.

$$D = \sum_{(f_i, f_j, \ldots, f_k)} p(f_i, f_j, \ldots, f_k) d_{i,j,\ldots,k} \tag{5.6}$$

Design diversity can be calculated through exhaustive simulation based on fault injection. Technique is proposed to efficiently estimate design diversity [134]. In [133] design diversity based design achieves significantly reliability against CMFs.

An example of design diversity is present in Fig. 5.16, where two implementations of 1-bit full adder and subtractor are shown. The calculated design diversity under the worst case condition [133] is present in Table 5.8. The results show that the duplex system with different implementations achieves better design diversity, which corresponds to higher reliability.

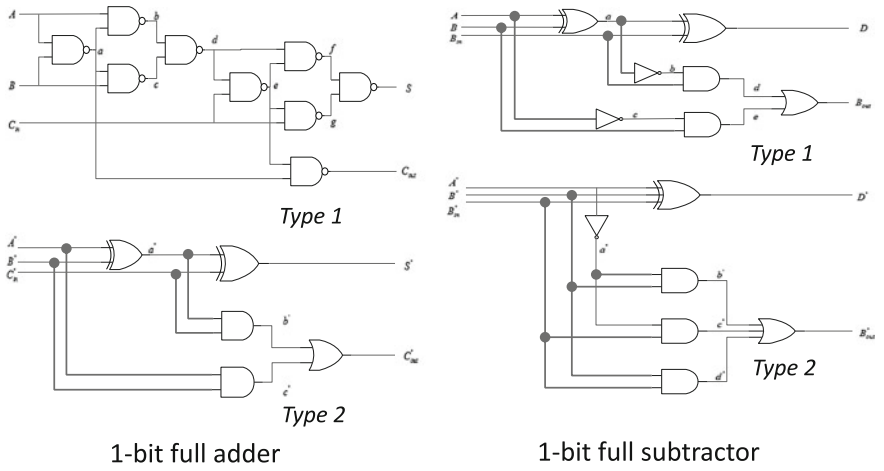


Fig. 5.16 Implementation for Full Adder (FA) and Full Subtractor (FS) [208] Copyright ©2015 **IEEE**

Table 5.8 Design diversity for different implementations in Fig. 5.16 [208] Copyright ©2015 **IEEE**

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

5.3.2 Graph-Based Diversity Analysis

Previously the design diversity metric is adopted to analyze circuit-level redundant techniques. In this work, it is applied for architectural analysis using a combined approach of graph-based analysis and circuit design diversity. The analysis on Major computational building blocks such as RISC and VLIW processors, as well as CGRA (Coarse-Grained Reconfigurable Architecture), are presented. The combined flow is briefed as following:

1. Quantify the amount of *conflict* functional units which can be *simultaneously* executed through graph based exclusiveness analysis.
2. Calculate circuit-level design diversity for the conflict functional units with the technique in Sect. 5.3.1.
3. Use the quantified design diversity to estimate application-level design diversity.

The proposed analysis flow estimates the ***maximal*** design diversity for the specific architecture, which can be used to evaluate reliability among architectures. The graph-based analysis is originated from graph representation of LISA operations, which is introduced in the following.

5.3.2.1 Graph Representation in LISA Language

LISA 2.0 language [1] has been used to describe various architecture variants such as ASIP [184], ASIC [200], CGRA [151]. The key concept of LISA is the Directed Acyclic Graph (DAG) of operations. A DAG can be represented as a graph $D =< V, E >$, where V indicates operations performing specific functions and E represents activation or scheduling of the child operations by the parental ones. Figure 5.17 visualizes the DAG for a RISC processor with 5 pipeline stages. In decode stage, 4 groups of operations are decoded into EX stage for execution. The DAG also shows the coding fields of specific operations, which are terminal field (shown as bit '0' or '1') or non-terminal fields (shown as label referring to child operations).

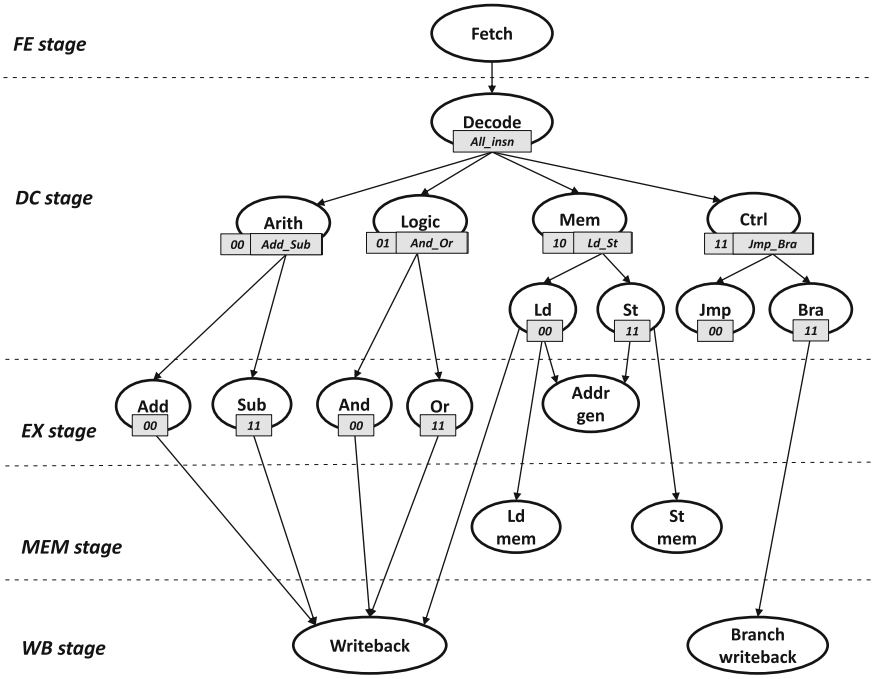


Fig. 5.17 Directed acyclic graph with ISA coding for ADL model [208] Copyright ©2015 **IEEE**

5.3.2.2 Exclusiveness Analysis

The exclusiveness analysis of operations determines whether operations in the DAG can be executed in the same clock cycle. It is originally proposed in [212] for decision on resource sharing of mutually exclusive operators. The information on exclusiveness can be extracted from the coding and activation condition in DAG, into another graph representation called *conflict graph* as shown in Fig. 5.18. The edges between operations in conflict graph indicate that they are *not* mutually exclusive or conflict, which can be executed in the same cycle. Besides, operations from different pipeline stages are shown in different colors and are always conflict with each other. For simplicity, edges between operations from different stages are ***not*** shown. An example is that operation Arith in Fig. 5.18 is conflicting with Decode, Add, Sub, And and Or, but exclusive with the rest operations.

5.3.2.3 Diversity Analysis

One desired requirement to the redundant system is the simultaneous execution of logic functions on duplicated hardware copies. Such information can be acquired

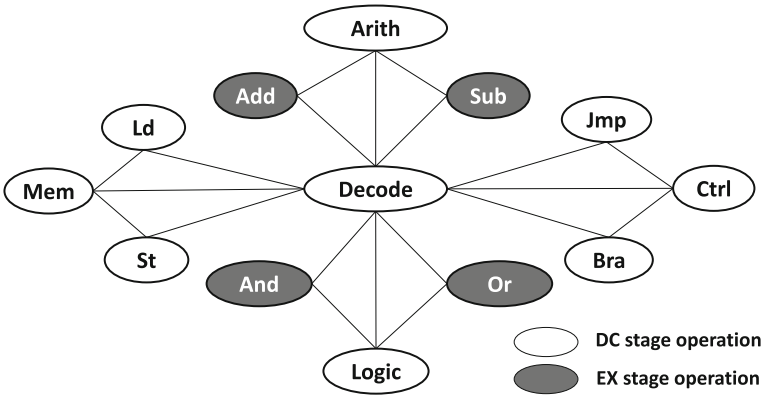


Fig. 5.18 Conflict graph for selected operations in Fig. 5.17 [208] Copyright ©2015 **IEEE**

from the exclusiveness analysis of the DAG graph. To incorporate the analysis, a novel graph representation named *Conflict Multiplex Graph* (CMG) is proposed with following information:

Theorem 5.3.1 *Exclusiveness is indicated by colors, where the operators with the same color are mutually exclusive.*

Theorem 5.3.2 *Functionality is indicated by edges, where the solid edge between operations indicates identical implementation and the dash edge shows diverse implementation.*

Figure 5.19 presents the CMG as well as the DAG for the EX stage of RISC processor, which consists of 7 operations. This work mainly focuses on the arithmetic and logical operations, which exist among all architecture variants. Compared to Fig. 5.17, 2 additional operations are decoded by the coding field *Chk*, which is intended to check both Arith and Logic operations. The operations decoded by *Chk* and *All_insn* are conflict with each other since they are from different coding fields in *Decode*. Hence, *MAC* and *And2* are shown in different colours as the rest ones. Regarding functionality, *MAC* can achieve the same functionality as *Add*, *Mul* and *Sub* with diverse implementations, so they are connected by dash edges. *And2* can only duplicate *And1* with identical implementation, which indicates a solid edge between them. No further edges exist in the CMG since other operations are either not able to repeat functionality or mutually exclusive.

The CMG based analysis assists to quantify the duplex/multiplex pairs for a specific operation, while the design diversity for each pair is calculated from circuit-level simulation technique as in Sect. 5.3.1. The calculated diversity for selected pair of logic functions is listed in Table 5.9.

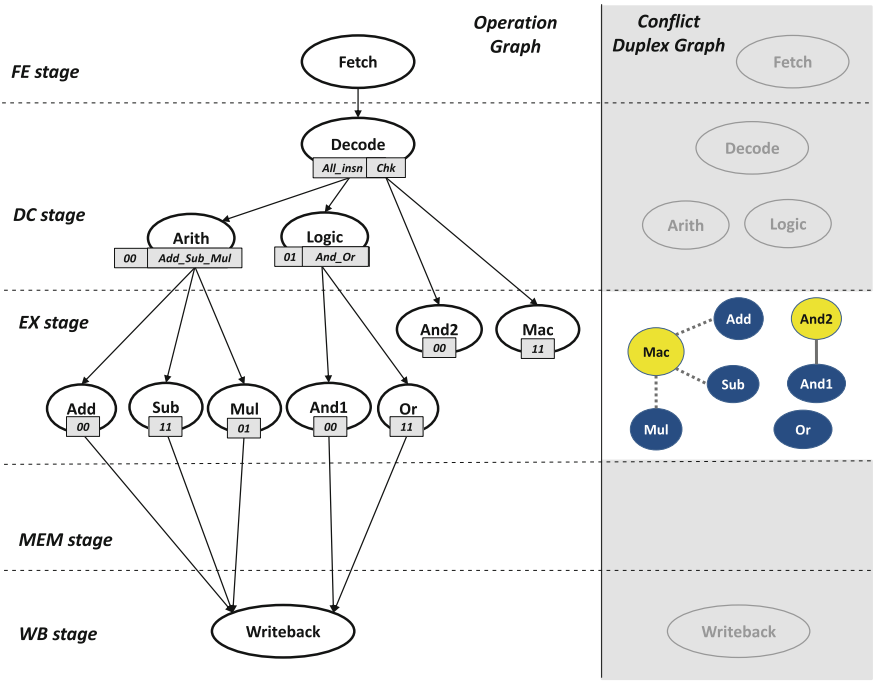


Fig. 5.19 Directed acyclic graph and conflict multiplex graph [208] Copyright ©2015 **IEEE**

Table 5.9 Duplex pairs for EX pipeline stage in Fig. 5.19 [208] Copyright ©2015 **IEEE**

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

5.3.2.4 CMG for Several Architecture Variants

In this section, CMGs of several architectures are presented to identify the redundancies in architecture level. It is worth noticing that such analysis detect the theoretical ***maximal*** redundancy. Further software or compilation techniques must be designed to actually utilize such redundancy, which is not covered in this work. The CMG-based analysis provides an analytical methodology to benchmark design diversity for different architectures.

TMR Triple Modular Redundancy (TMR) is a widely used technique which exploits three logic units to verify the correctness of protected operation. One example of the CMG of TMR architecture is shown in Fig. 5.20, where *Add* and *Sub* operations are

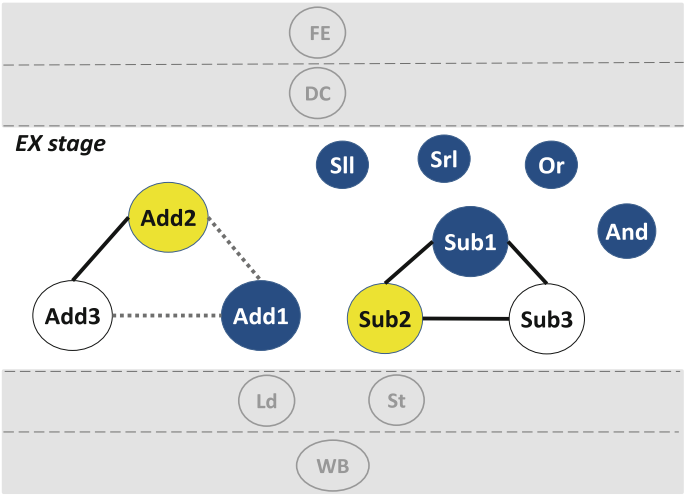


Fig. 5.20 Conflict multiplex graph for TMR Architecture [208] Copyright ©2015 **IEEE**

under protection by two extra copies. *Add2* and *Add3* are identical while *Add1* is diversely implemented. All *Sub1*, *Sub2* and *Sub3* are identical. The diversity of such multiplex pairs is calculated by Eq. 5.6. It is worth mentioning that *Add2* and *Add3* are conflict with all operators in blue colour. However, the regular TMR implementation, which groups all three addition operations together without access to others, may limit *Add2* and *Add3* to form pairs with operations other than *Add1*.

URISC URISC [150] proposes the fault tolerance technique by adopting the Turing complete instruction *subleq*, which executes in the co-processor to diversely duplicate the instructions in the main processor. The approach is abstractly present in Fig. 5.21. Since *subleq* is separately decoded in the coprocessor and able to perform functionalities of all operations, it forms the diverse pair with all operations in the main core.

VLIW Multiple instruction syllables, which are separately decoded, are applied in VLIW processor for parallel execution. The CMG for VLIW with four syllables are present in Fig. 5.22. Each operation in one syllable are conflict with all operations from other syllables to form multiplex system. For example, *Sub1* can form identical duplex pair with *Sub2*, *Sub3* and *Sub4*, while also diverse pair with *Add2*, *Add3* and *Add4*.

CGRA CGRA architecture consists a large number of processing tiles interconnected through a specific network topology. Several prefabricated functional units (FUs) exist in each processing tile, whose functionalities are selected during the post-fabricated configuration phase. The difference between CGRA and FPGA is that FPGA applies the FU of look-up-table, which can realize fine-grained design than CGRA. For each configuration, only one function is realized in each tile, which

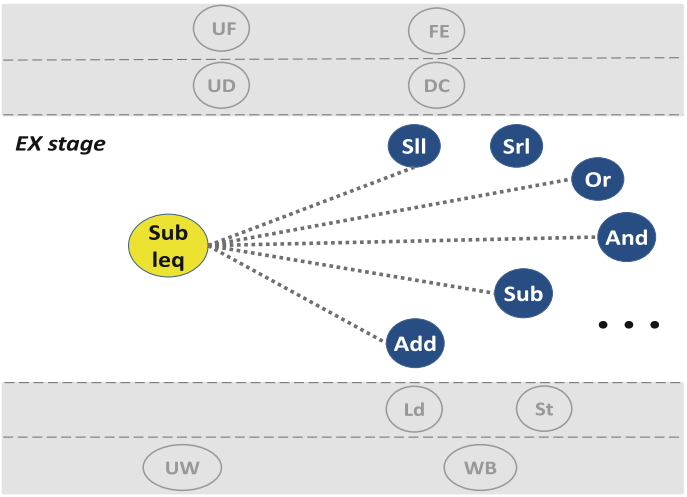


Fig. 5.21 Conflict multiplex graph for URISC Architecture [208] Copyright ©2015 **IEEE**

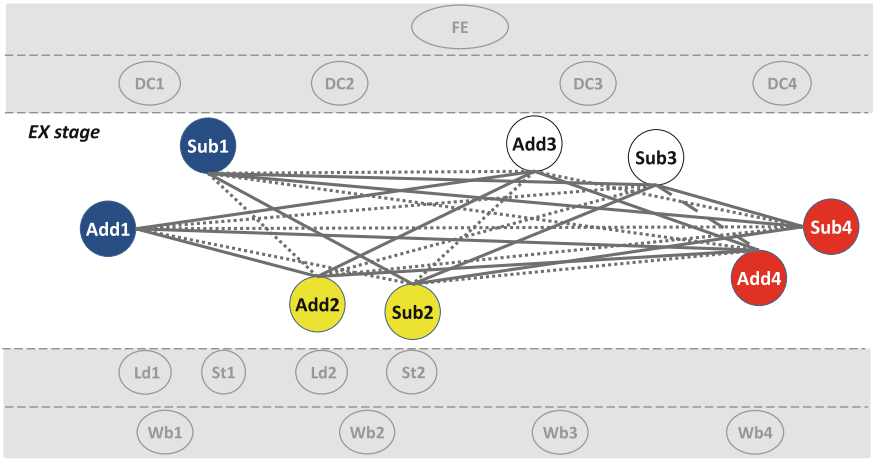


Fig. 5.22 Conflict multiplex graph for VLIW Architecture [208] Copyright ©2015 **IEEE**

shows FUs inside one tile are mutually exclusive. However, the configuration does not constrain the FU functionality across tiles. The Fig. 5.23 shows the CMG of CGRA with six tiles, where a huge amount of identical and diverse pairs are indicated.

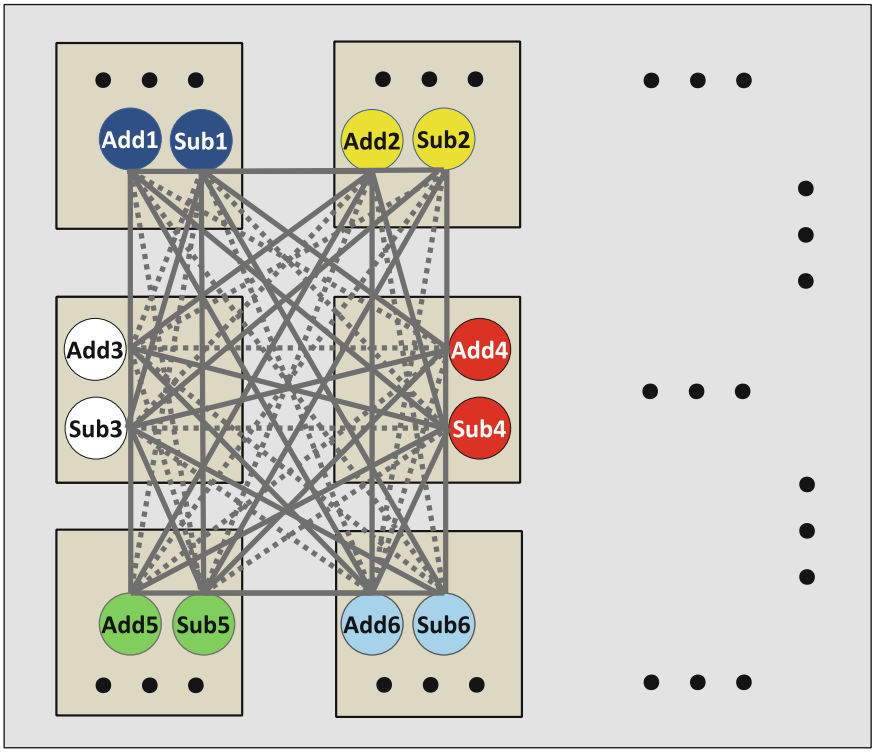


Fig. 5.23 Conflict multiplex graph for CGRA Architecture [208] Copyright ©2015 **IEEE**

5.3.3 Results in Diversity Estimation

This section presents several case studies on design diversity based reliability analysis. First, application-level design diversity is estimated based on architecture-level design diversity and instruction statistics. After that, system-level Mean-Time-To-Failure (MTTF) is derived from design diversity.

5.3.3.1 Architecture Diversity Evaluation

Three architecture variants including RISC, VLIW and CGRA are present for analysis. Four exemplary operations, which are *Add*, *Sub*, *Sll*, *Srl*, are chosen for calculation of design diversity. Table 5.10 lists the number of pairs for both identical and diverse system. The identical system consists of a single type of modules for each operation, while the diverse system has an equal number of two types of modules. Design diversity is evaluated according to Eq. 5.6, where all modules of the same operation are used to verify the correctness of such operation.

Table 5.10 Architecture variants of design diversity evaluation [208] Copyright ©2015 **IEEE**

	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

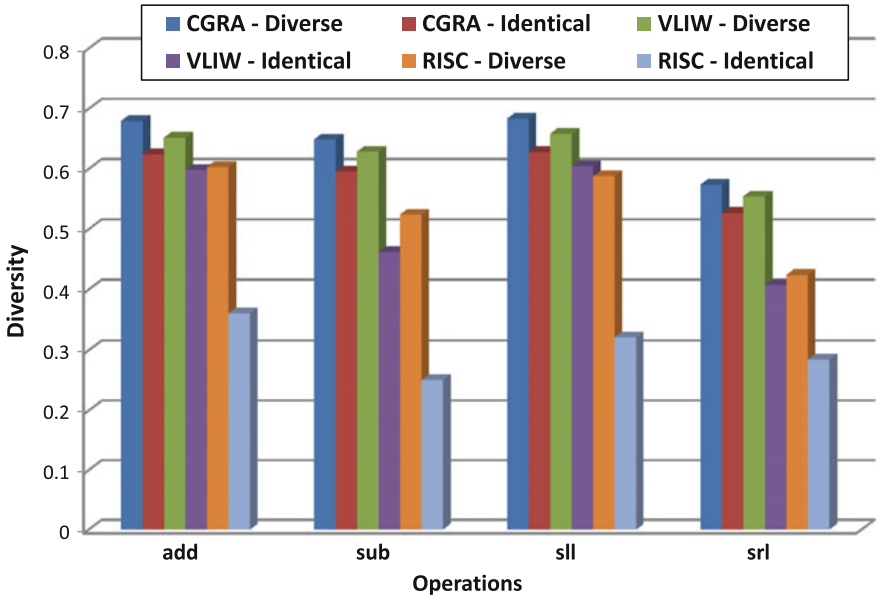


Fig. 5.24 Design diversity of architecture variants [208] Copyright ©2015 **IEEE**

Figure 5.24 shows the estimated architecture-level design diversity, which shows similar trends among all architectures. More modules in the system always achieve higher design diversity. With the same amount of modules, diverse implementations lead to better design diversity than identical ones. Quantitatively speaking, RISC architecture with two diverse modules has comparable design diversity as VLIW with four identical modules.

5.3.3.2 Application-level Diversity Evaluation

Taking advantage of architecture-level analysis, application-level design diversity is introduced in Eq. 5.7. While D_{op} directly refers to the architecture design diversity

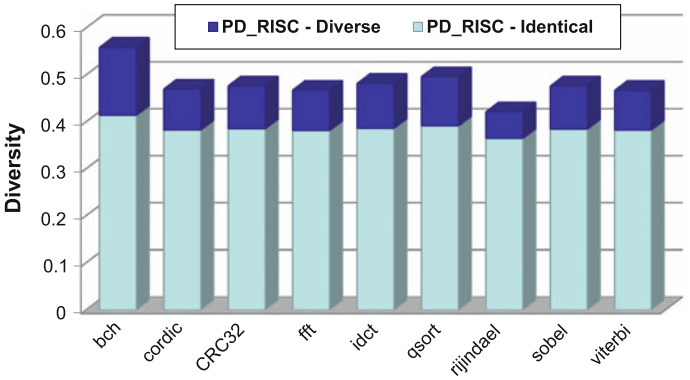


Fig. 5.25 Application-level design diversity for PD_RISC processor [208] Copyright ©2015 **IEEE**

for operation *op*, $P_{op,app}$ is the percentage of operation *op* among all operators in application *app*. Assembly-level instruction profiler can find $P_{op,app}$ for any high-level applications. To increase application-level design diversity and reduce error probability, it is desirable to execute the operations with a higher percentage on more diverse modules.

$$D_{app} = \sum_{op} P_{op,app} D_{op} \tag{5.7}$$

The PD_RISC processor from the IPs of Synopsys Processor Designer [184] is used to evaluate design diversity for several embedded applications. The cycle-accurate instruction-set simulator generates the statistics on instruction profiling.

Figure 5.25 presents the evaluation of application-level design diversity on PD_RISC processor. `Add`, `Sub`, `Sll`, `Srl` are targeted operations. Among all applications, diverse systems result in higher design diversity than identical ones. The difference in absolute values is caused by the difference in operation percentage among applications.

5.3.3.3 Mean-Time-To-Failure Estimation

$MTTF_{op}^{arch}$ for a specific operation *op* of the architecture *arch* can be estimated using the failure rate λ_{op}^{arch} introduced in Eq. 5.8. For a transient bit-flip fault model, by Eq. 5.9, λ_{op}^{arch} is further derived from $P_{op}^{1fault,arch}$, which is the probability of one 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 Eq. 5.5 and D_{op}^{arch} is the design diversity of multiplex system with operator *op* in architecture *arch*. In Eq. 5.10, $P_{op}^{1fault,arch}$ is further related to the architecture dependent product of module-level

Table 5.11 Failure rate estimation for four operators [208] Copyright ©2015 **IEEE**

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

fault probability $P_{op,i}^{1fault}$, which corresponds to the division of area estimation of the operator $A_{op,i}$ by the constant $A_{1fault/hour}$. $A_{1fault/hour}$ is the size of area that injection of one fault happens per hour under a specific environmental condition. Such condition is acquired by the reciprocal of Failure-in-Time (FIT) [71] in Eq. 5.11. For instance, this work assumes the FIT as $10^{-4} cph/\mu m^2$. The unit is fault count per hour (cph) per unit area (μm^2).

$$MTTF_{op}^{arch} = \frac{1}{\lambda_{op}^{arch}} \tag{5.8}$$

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

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

$$A_{1fault/hour} = \frac{1}{FIT} \tag{5.11}$$

Table 5.11 presents estimated $A_{op,i}$ and $P_{op,i}^{1fault}$ for four operators according to information of 90nm Faraday technology cells [61].

Calculated by D_{op} from Fig. 5.24, the estimated $MTTF_{op}^{arch}$ for four operators on several architecture variants is present under logarithmic scale in Fig. 5.26. It is observed that CGRA architecture is naturally more robust than VLIW, which is on the other hand reliable than RISC architecture.

MTTF increases with both the increasing number of modules in the system and the design diversity for the same operation. Benchmarked with Fig. 5.24, RISC architecture with two diverse modules leads to significantly less *MTTF* than VLIW with four identical modules. This is caused by the fact that P_{op}^{1fault} for VLIW is much smaller than RISC since more modules in the multiplex system indicate a lower probability that one single type of fault happens in ***each*** module. CGRA shows similar trends as VLIW. Among all four operators, `Sll` shows the highest *MTTF* which results from its smallest size and relatively high design diversity.

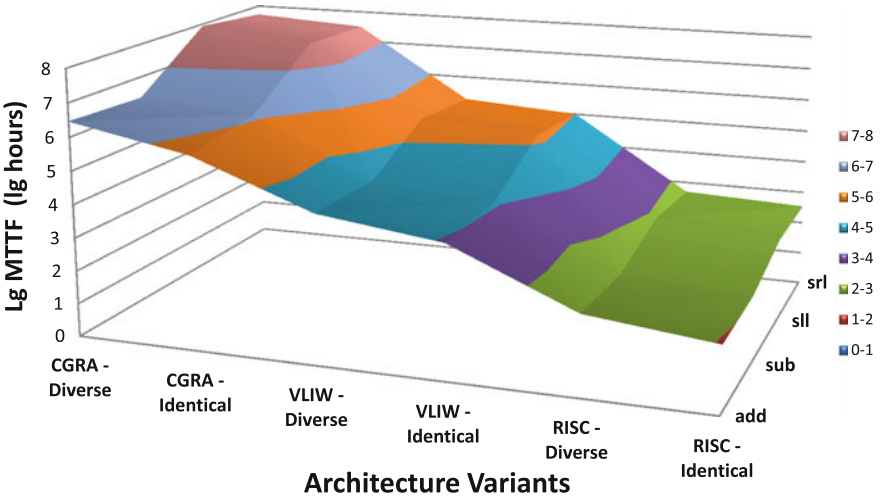


Fig. 5.26 Mean-time-to-failure of architecture variants [208] Copyright ©2015 **IEEE**

5.3.4 Summary

In this work, design diversity metric, which is originally proposed to quantify reliability for circuit-level designs, is extended into the architecture-level analysis of different processing architectures. This is achieved through a novel graph-based analysis on functionalities and exclusiveness of operations in the architecture. The proposed approach is applied to architecture and application-level design diversity estimation, as well as system Mean-Time-To-Failure.

Chapter 6
Architectural Reliability Exploration

In this chapter, three architecture-level fault tolerant techniques are presented. In Sect. 6.1 opportunistic redundancy is proposed to protect the algorithmic units of embedded processor with a low performance penalty. In Sect. 6.2 asymmetric redundancy is proposed to protect the memory elements with the feature of unequal error protection based on information criticality. In Sect. 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

The architecture-level fault tolerant techniques for mainstream processors can be classified into temporal and spatial redundancies. Temporal redundancy achieves protection through replicated execution of the instructions on the same hardware units, such as Simultaneous and Redundantly Threaded (SRT) processor [157]. Spatial redundancy relies on additional hardware units such as Error Correcting Code (ECC) [115] for memory protection or Triple Modular Redundancy (TMR) [198] for logic protection.

By contrast, the conventional fault tolerant techniques are less explored for embedded processors. Spatial redundancy is limited by tight resource and power constraints, whereas real-time constraint prohibits the exploration of temporal redundancy. However, redundancy within embedded processors has been never completely eliminated nor fully explored for reliability enhancement, which is caused by the complexity of compiler and dynamic hardware usage. Consequently, the underutilized resources in the processor such as execution units are advised to be *opportunistically* used (protect when it is possible) for best-effort reliability enhancement. Furthermore, duplication can be performed at micro-architecture-level instead of instruction-level, in order to reduce performance penalty.

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_6

Contribution This work proposes low-cost fault tolerant techniques based on the concept of opportunistic redundancy. Two protection policies are introduced. The aggressive policy always replicates protected operations to ensure computational correctness. The passive policy performs re-execution only when underutilized resources are detected. Relatively large performance penalty is incurred by aggressive policy while passive policy achieves zero penalties. The proposed protection schemes are demonstrated on customized RISC and VLIW processors with novel fault tolerant features. Fault injection technique proposed in Chap. 4 is used for reliability evaluation.

6.1.1 Opportunistic Protection

In this section, the concept of opportunistic redundancy is introduced by illustrating on an embedded RISC processor. The Directed Acyclic Graph (DAG) of the processor is present with the focus on the protected units. Both protection policies are explained after that.

6.1.1.1 Processor Modeling

The exemplary processor is designed by ADL LISA [1] with a DAG graph in Fig. 6.1, which represents the coding and scheduling of processor operations. The operations are associated with processor pipeline stages to represent the scheduling information. The directed edges between operations represent the activation information. For more details on LISA language, the audiences are referred to [1].

The DAG graph shows different groups of instructions are decoded from the *Decode* operation. The *alu* and *bit* directly provide operands for the *alu_ex* in execution stage, which implies the intra-instruction dependency. Besides, the *control*, *cmp*, and *mem* instructions do not provide operands for *alu_ex* but use its output operands from previous clock cycles. Such inter-instruction dependency is labelled by the dash lines.

6.1.1.2 Protection Level

The level of protection for a fault tolerant design influences the fault coverage and performance penalty. For instance, the RMT processor relies on instruction-level protection, which incurs performance loss for each duplicated instruction. The proposed approach realizes the protection into the micro-architecture level. As shown in Fig. 6.1, the *alu_ex* operation is the key for the success of both data and control flows of the processor, which indicates its significance for protection. Furthermore, the importance of *alu_ex* can be shown from instruction profiling of typical benchmarks. Figure 6.2 presents the average instruction percentage among 12 applications

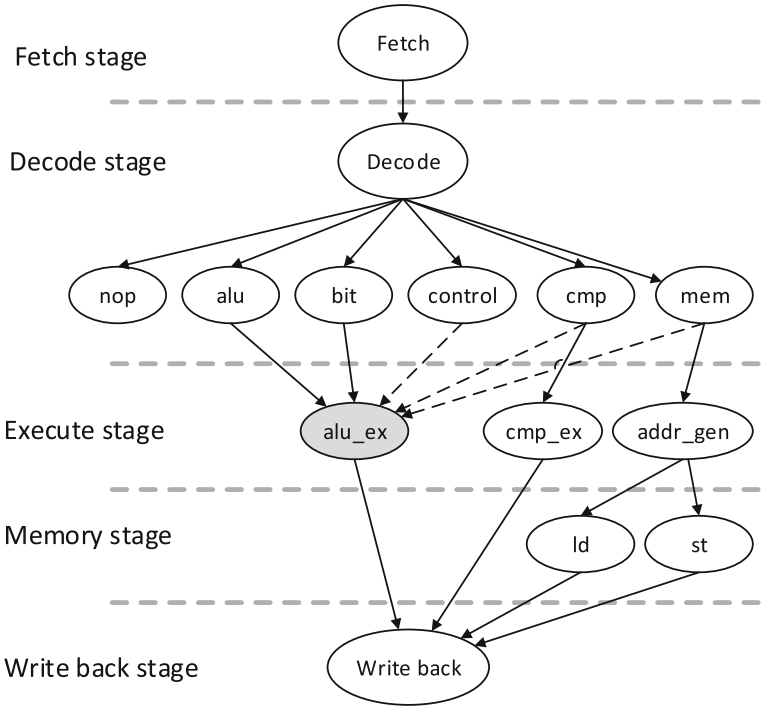


Fig. 6.1 Directed acyclic graph of embedded RISC processor [203] Copyright ©2013 **IEEE**

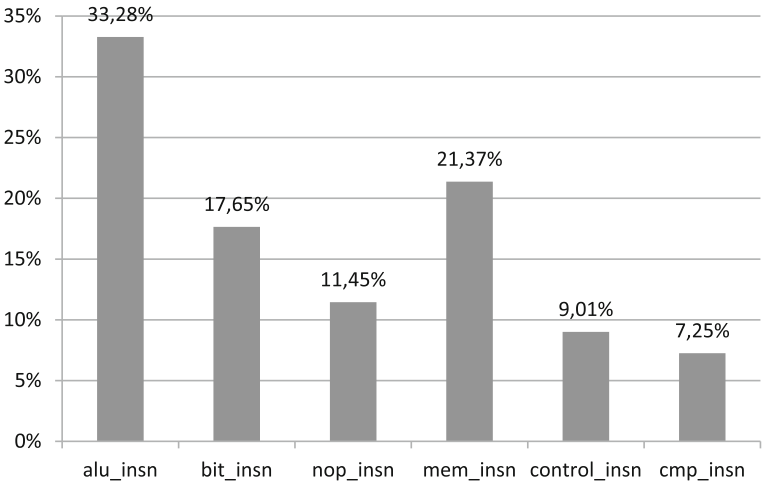


Fig. 6.2 Average instruction distribution for MiBench [203] Copyright ©2013 **IEEE**

of the MiBench suite [73]. It is observed that *alu* and *bit* contributes 51% of all the instructions whereas *compare*, *memory*, and *control* account for another 37%. The NOP instructions are less relevant for protection since it is mostly for the scheduling purpose without any functionality.

As a result, the operations in micro-architecture unit *alu_ex* are required for re-execution with the same operands to ensure the correctness. The desired architecture level technique for opportunistic redundancy keeps detecting idle cycles of *alu_ex*, where instructions other than *alu* and *bit* are in parallel performed in the Execute pipeline stage. Previous operations are then scheduled on *alu_ex* with the same operands stored in the protection buffers. Detected result mismatches activate the roll-back of the processor states. Compared to instruction-level duplication, the proposed technique achieves protection without affecting the performance of main instruction flow when no errors are detected.

6.1.1.3 Protection Policies

The protection policies are explained through different flows of assembly instructions in Fig. 6.3. Figure 6.3a indicates the duplication of *add* instruction at line 3 of NOP. An improved protection in Fig. 6.3b can duplicate last two ALU instructions when consecutive NOPs are detected. For such purpose, buffers to store the last two instructions are required. Policy-directed decisions have to be made when there are not enough NOPs or the amount of instructions in protection buffer exceeds the buffer size. Figure 6.3c indicates the passive policy where the oldest instruction (*add* from line 2) is ignored. The aggressive policy in Fig. 6.3d forces the pipeline to stall in order to win one extra cycle for protecting the *add* from line 2. Other than temporal redundancy, spatial redundancy can also be exploited for VLIW processor. Shown as in Fig. 6.3e, the VLIW processor with four syllables can duplicate instruction on either a different syllable of the current instruction or syllables of the next instruction.

6.1.2 Implementation

The proposed techniques are demonstrated on embedded RISC and VLIW processors. In this section, we discuss the details of implementation on both architectures to achieve opportunistic redundancy.

6.1.2.1 RISC Model

PD_RISC_32p6 processor is a 6-pipeline stage fully bypassed RISC IP from Synopsys Processor Designer. Features including C compiler, instruction-set simulator, and RTL generation are supported for processor customization.

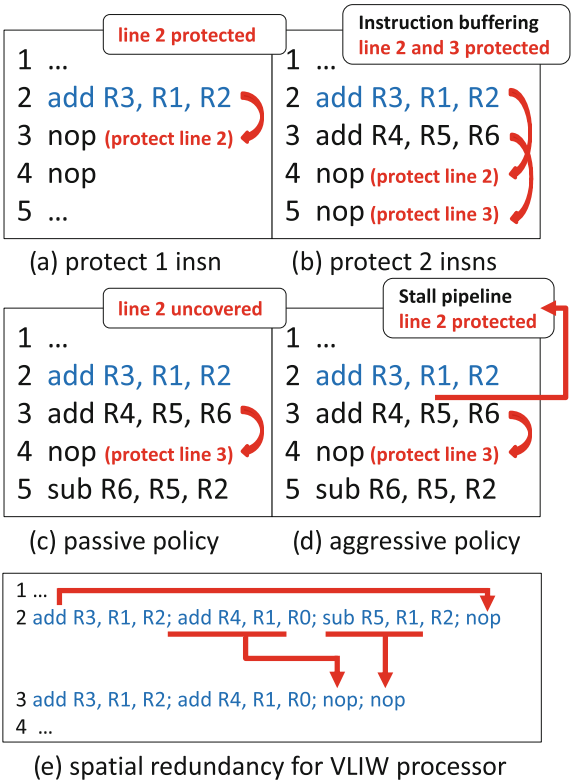


Fig. 6.3 Protection policies for RISC and VLIW processors [203] Copyright ©2013 **IEEE**

Protection unit The protection unit locates in the EX pipeline stage as the centralized controller of the protection flow. Figure 6.4 visualizes the states of protection unit. At each clock cycle, the state machine of protection unit checks the availability of ALU (*alu_ex* operation in Fig. 6.1). The idle ALU units will activate the protection state by fetching the oldest entry in the protection buffer for re-execution in ALU. When the repeated execution matches the previous one, such instruction is committed from the commit buffer together with the instruction depending on it. On the other hand, a result mismatch triggers error recovery by flushing the pipeline and roll back the previous state of the program counter.

When the ALU is occupied, decisions have to be made on whether to put instructions into the protection and commit buffer. The instructions are classified as hard and soft based on instruction type. The instruction executed in the ALU is considered as HARD type while the other instruction depends on the ALU one is considered as SOFT type as labeled in Table 6.1. HARD instructions are sent into protection buffer for duplicated execution, whereas the SOFT instructions are moved into the commit buffer waiting for the check of their dependent instructions. The HARD instructions are opportunistically executed according to the protection policies as

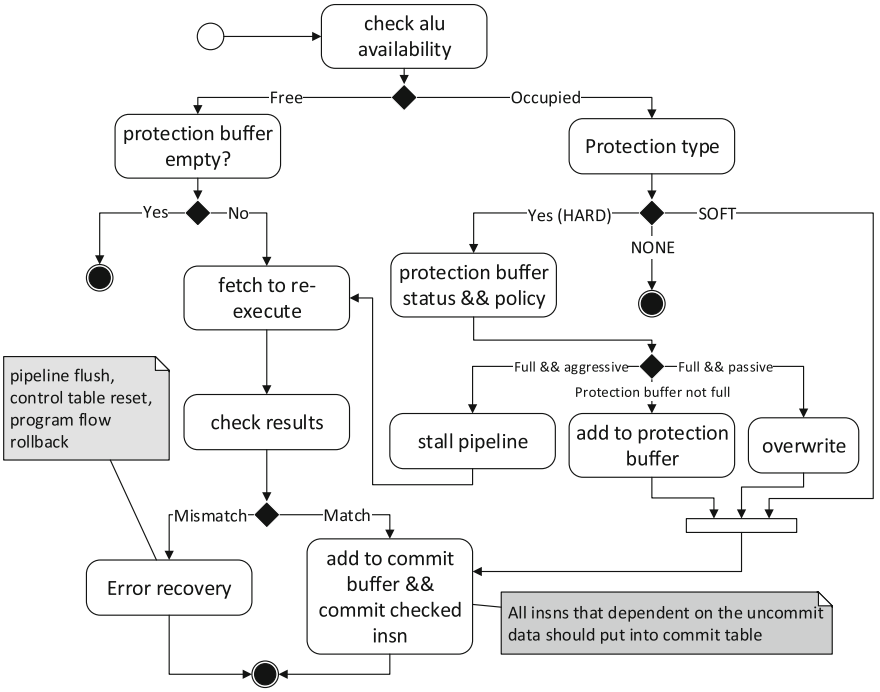


Fig. 6.4 Execution flow of the protection unit [203] Copyright ©2013 **IEEE**

Table 6.1 Handling methods of different instruction types [203] Copyright ©2013 **IEEE**

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

shown in Fig. 6.3. The instructions in the protection buffer are also duplicated in the commit buffer to maintain correct instruction flow.

Figure 6.5 visualizes the customized processor with protected ALU unit in EX pipeline stage. In the following architecture units for protection purpose are introduced.

Protection buffer The opcode, operands, and result values are kept in the entry of protection buffer as well as an additional checkState tag to indicate whether the instruction has been verified. The program counter (PC) of the instruction is also maintained when roll-back is encountered. The buffer is arranged as a circular buffer where new entry erases the oldest entry when the buffer is full. During instruction set simulation, the buffer size is adjustable to trade-off reliability with performance and physical overheads. The decoded instructions from DC pipeline stage activate the protection unit to manage the protection buffer.

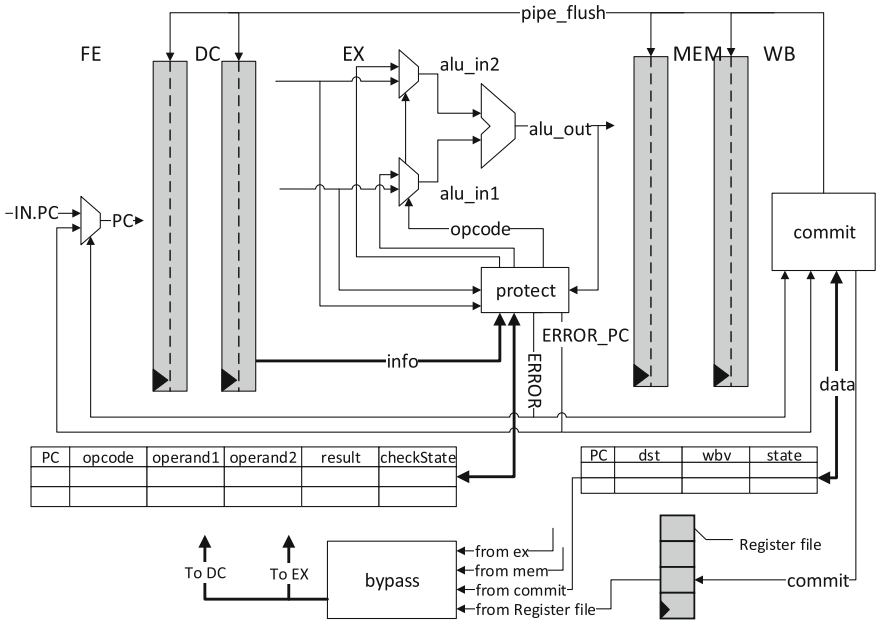


Fig. 6.5 RISC architecture with protected ALU unit [203] Copyright ©2013 **IEEE**

Commit buffer The commit buffer keeps the PC of dependent instruction, the return values (wbv) and their addresses (dst) for both HARD and SOFT instructions. Instruction after protection commits itself and dependent instructions into the register file and data memories according to the matching of PC values. Since the RISC processor has fully bypass features, a new path of bypassing is established from the commit buffer to the DC and EX pipeline stages. Such extra path maintains the correct data flow when the operands in the commit buffer are demanded in the pipeline. Bypassing from commit buffer has higher priority than that from the register file to ensure that correct operands are fetched.

Error correction A detected mismatch during repeated execution activates the rollback procedure by fetching previous instructions based on the PC of the erroneous instruction in the protection buffer. Both buffers and pipeline registers are flushed.

6.1.2.2 VLIW Model

The LT_VLIW_32p5x4 processor consists 5 pipeline stages with 4 parallel syllables. The EX pipeline stage in each syllable has an individual ALU unit. Due to the compiler complexity and data dependency in the application code, several syllables are usually idle which offers the spatial redundancy for protection. A block diagram of the customized architecture is present in Fig. 6.6. The *operand routing* unit is

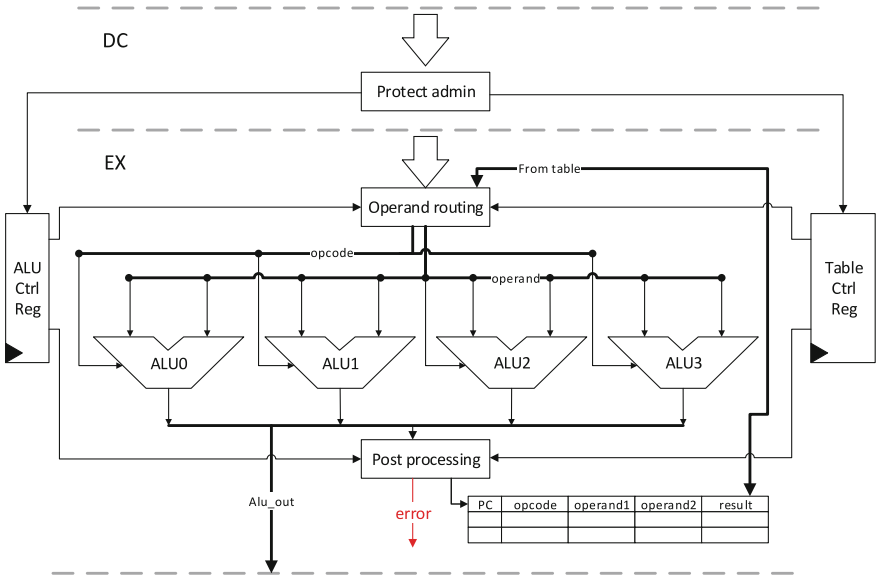


Fig. 6.6 VLIW architecture supporting opportunistic redundancy [203] Copyright ©2013 **IEEE**

designed for dynamically routing the operands to the idle ALUs, while the *post processing* unit compares the result from different executions to generate error signal. The *protect admin* and *ALU control register* work together to ensure the distribution of protected operations. The protection table (buffer) and control registers are similar to the ones for the RISC architecture.

ALU control register The 16-bit *alu_ctrl_reg* register consists of four fields to control the protection state of four ALU units. As shown in Table 6.2, the two MSB bits of each field indicate the operation mode, which configures the source and destination of the ALU operation. The two LSB bits of the each implement a pointer targeting corresponding locations in the protection table or the syllables. The value of *alu_ctrl_reg* is set by the *Protection admin* unit in the decode pipeline stage. Figure 6.7 presents an example on the behavior of each ALU directed by the control register. Occupied entries in syllables and protection table are shown in dark color. For instance, *alu0* and *alu1* are occupied by ALU instructions as well as the first entry in protection table. Based on the values in control register, the idling *alu2*

Table 6.2 ALU control register [203] Copyright ©2013 **IEEE**

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

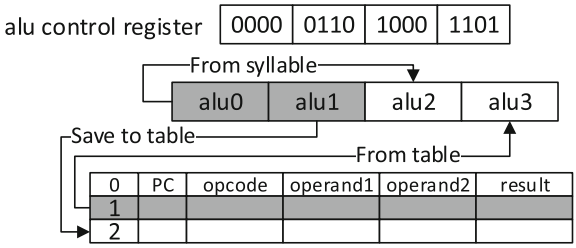


Fig. 6.7 VLIW control register [203] Copyright ©2013 **IEEE**

duplicates the operation in *alu0*, while *alu3* re-executes the operation in first entry of protection table. operation in *alu1* is moved into the protection table entry two to wait for chances of re-execution. When the table is full, the decision on allocation of new entries has to be made according to the protection policies.

Commit unit The delayed commit unit is not required for VLIW architecture since the instructions are either duplicated or overwritten, which is caused by the internal parallelism of the architecture. Detailed timing analysis is omitted in this section for the sake of simplicity.

6.1.3 Experimental Results

Experimental results on both architectures are present in this section. First, performance overheads are investigated for different benchmarks according to proposed policies. After that, reliability enhancement is estimated based on the fault injection experiments. Lastly, the physical overheads are present for the proposed design.

6.1.3.1 Performance Overhead

Several benchmarks from the MiBench are ported to be executed on the proposed architectures. Figure 6.8 presents the coverage of instructions on PD_RISC under both protection policies. In the experiment, both protection and commit buffers are configured to store three entries. It is observed that the aggressive policy covers significantly more instructions than the passive policy. Such high coverage comes at the cost of huge performance degradation during the stalling clock cycles. However, the aggressive policy still incurs less degradation than the approach of RMT processor, which is 100% due to instruction-level duplication. Moreover, the passive policy achieves decent protection and does not incur any degradation, which exploits the redundancy optimally. Compared to different benchmarks, the computational intensive application such as *rijndael* and *sha* incur larger degradation due to the shortage of idle ALU resources. The VLIW profiling shows similar instruction coverage as the RISC, which is not present for simplicity.

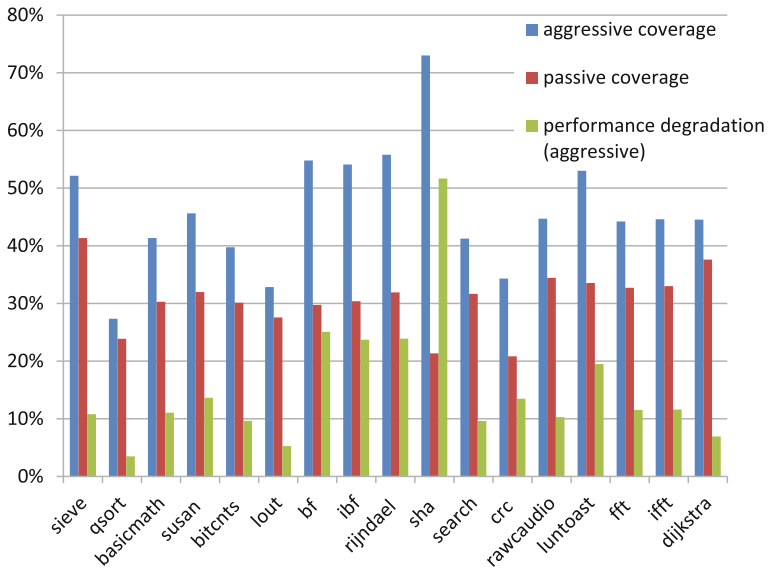


Fig. 6.8 Instruction coverages and performance degradation on RISC [203] Copyright ©2013 **IEEE**

6.1.3.2 Fault Injection

Fault injection experiments Sect. 4.1 are used to demonstrate the effectiveness of proposed techniques. Single Bit-flip faults are randomly injected into the ALU units of processors during instruction set simulation. *Error Manifestation Rate* (EMR) [44] is adopted for error evaluation, which indicates the percentage of detected error on the memory interface to the processor core. For more on EMR metric, the audiences are referred to Sect. 4.1.

10,000 experiments are conducted to acquire the statistical EMR value under a fixed amount of faults. *Sieve* and *IDCT* benchmarks are compiled to run on the processors. Figure 6.9 shows the trends of EMR with increased faults for both RISC and VLIW processors.

For both architectures, protection under passive policies give significantly lower EMR values compared with the unprotected ones. The EMR under aggressive policy reaches zero for both applications regardless of a number of faults. This is caused by the fact that aggressive policy provides an always correct result. Between different architectures, VLIW produces fewer errors under the same amount of injected faults. It happens since VLIW spreads the faults into four ALU units, which reduce the probability of error on each ALU.

The proposed technique is influenced by the codes generated by the C compiler. For instance, different optimization flags generate various assembly codes, which affects the opportunities of protection for proposed techniques. The effects of compilation modes are investigated for CoSy compiler system [188] and shown in Fig. 6.10.

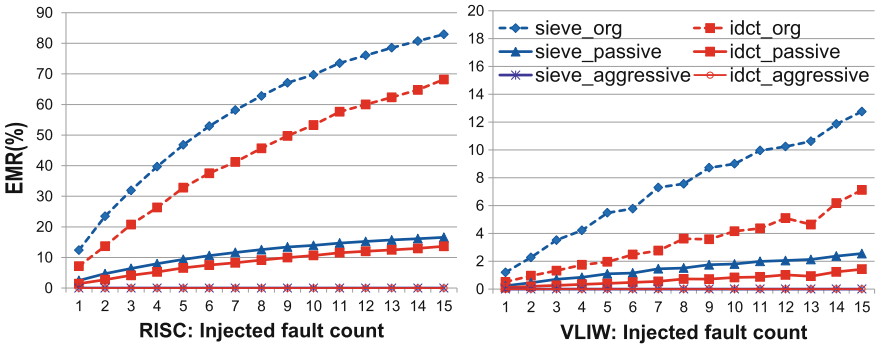


Fig. 6.9 EMR with increased count of faults for RISC/VLIW processor [203] Copyright ©2013 **IEEE**

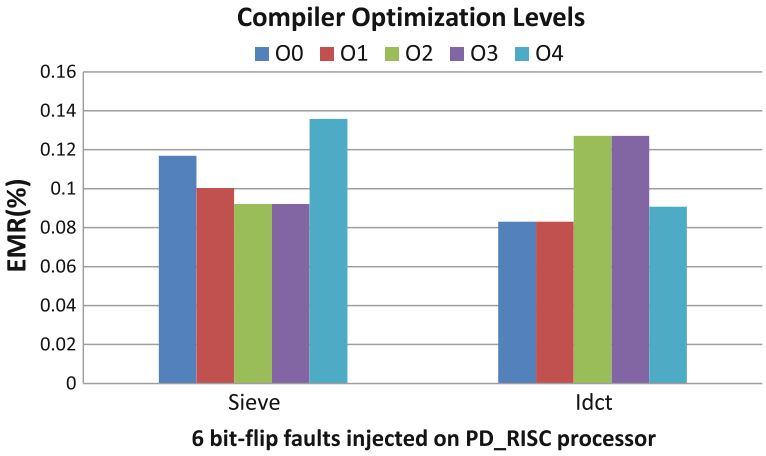


Fig. 6.10 Effects of C compiler optimization levels on EMR for passive mode [203] Copyright ©2013 **IEEE**

Features of optimization levels are briefly explained in the following where details on the compilation are referred in [145].

- O0: No optimization, as applied in Figs. 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

The difference in EMR results shows the impact of compilation techniques. Research in optimization techniques based on opportunistic redundancy is envisioned to trade-off reliability with the performance penalty.

Table 6.3 Design overheads for proposed architectures [203] Copyright ©2013 **IEEE**

	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%+

6.1.3.3 Physical Overhead

The proposed architectures are synthesized under 90 nm Faraday technology library by Synopsys Design Compiler. The estimated physical overheads (area and power) are present in Table 6.3. The protection and commit buffers are major contributors to the overheads since they are implemented as registers with specific control logic. Architecture only supporting error detection does not require the commit buffer which gives less overhead for the RISC. VLIW does not contain the commit buffer which gives the same area overhead for both detection and recovery modes. Operand routing and administration logic add more overhead to the VLIW architecture than the RISC. The energy overhead is estimated by the power overhead and performance penalties, which varies among different applications.

6.1.4 Summary

This work presents a best effort design methodology to increase the reliability of embedded processors. The idle states of ALU units are opportunistically exploited to duplicate previous instructions. Aggressive policy achieves full protection with performance penalty whereas passive policy performs protection only when the ALUs are idle. Novel features in the architecture of RISC and VLIW processors are implemented to realize proposed techniques. The effectiveness of proposed approach is demonstrated through fault injection experiments.

6.2 Asymmetric Reliability

State-of-the-art research on the design of reliable system shows two trends. First, reliability is recognized as a *cross-layer* design issue [47]. It claims that fault tolerant techniques on individual design abstractions can result in an over-designed system. In order to avoid over-protection, understanding of different design abstractions needs to be clearly established. The second trend is to offer *asymmetric reliability* to

unequally protect different parts of the system according to their significances [74]. Such criticality can be statically or dynamically assigned from system-level designer and the applications.

Although the researchers claim that asymmetric reliability techniques lead to more design trade-offs [74, 99, 111], it is still yet to be explored for processor resources, especially storages including instruction and data memories. For this aim, instruction vulnerability analysis has been proposed in [163] to address asymmetric software mapping. Unequal protection for register file has been proposed in [111] by support from compilation techniques. However, existing works does not systematically address the scope and impact of architecture-level asymmetric reliability techniques to our best knowledge.

Traditionally, processor storages are treated as noisy communication channels which are protected through information redundancy techniques such as Error Correction Code (ECC) [27]. Though rich literature in channel coding aids the work, two important design challenges need to be solved. First, the tight power and timing budgets in embedded domain demand for efficient implementation of channel encoder and decoder. A few works [95, 159] focus on applying Hamming codes for storages without in-depth consideration of the overhead issues. Second, asymmetry targeting reliability can be viewed from different perspectives. For instance, it is advised that different instructions should be unequally treated according to their impacts. However, the traditional view of asymmetry [105, 158] assigns unequal error probabilities on different bit positions of the message but with the same error distributions across all messages.

Contribution In this work, an asymmetric reliability model for processors is proposed which considers asymmetry from the perspective of both message types and execution conditions. Novel coding schemes are proposed to realize asymmetric reliability for processor resources. Customized architectures are developed to explore reliability trade-off with other design metrics. Detailed fault injection experiments are performed to verify the effectiveness of proposed coding scheme and architectures.

6.2.1 Asymmetric Reliability

A classification of schemes for asymmetric reliability is proposed in Fig. 6.11. The unequal protection for architecture resources can be either static or dynamic from the perspective of execution. The static approach fixes the protection approach before execution while the dynamic one alters the approaches according to the updates of criticality during runtime. From the perspective of message granularity, unequal protection can be performed for different bits of a message or among different messages. For instance, different bit positions of a data message have different significance. Errors in MSB usually gives higher failing probability of the system. However, such bit-wise asymmetry is less relevant for instruction words, where different types of

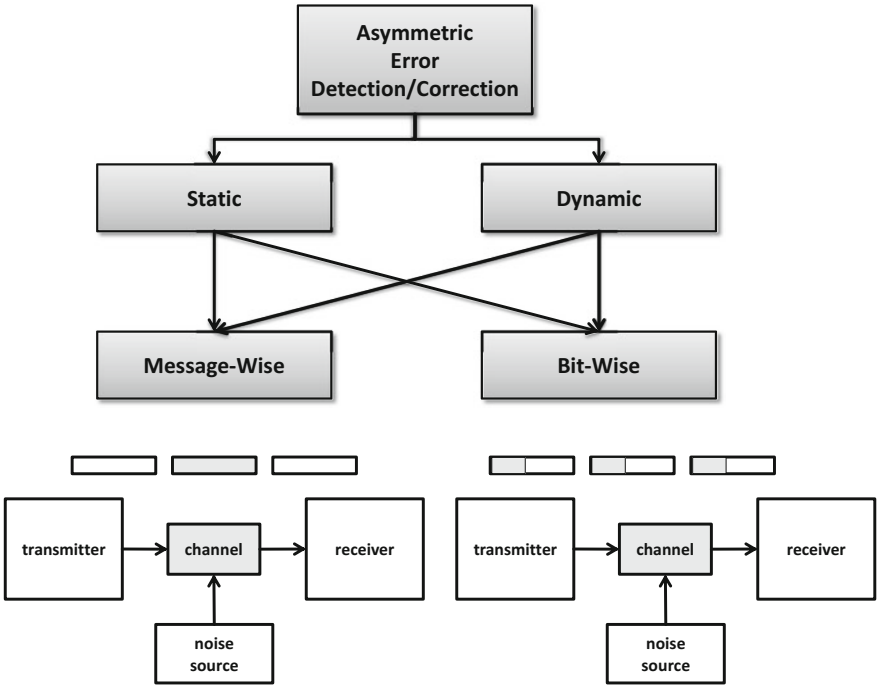


Fig. 6.11 Asymmetric encoding and decoding [205] Copyright ©2014 **IEEE**

instructions impact system execution to various extents. In general, control instructions require stronger protection than the arithmetic ones.

One possibility to apply the proposed schemes for processor resources is to implement different levels of protection through asymmetric coding techniques. Explicitly, the variants of linear codes have been widely adopted in communication technologies for forward error correction, which can be also customized for the processors. In the next, fundamentals on linear codes are introduced, followed up with an asymmetric coding scheme.

6.2.1.1 Linear Code Fundamentals

Any practical communication channel is exposed to several sources of noise which incur errors in the transmitted messages. The seminar paper of Shannon [27] established the foundation of the theory on ECC to deal with the issues of error detection and correction. Basically, before the transmission, a message m is transformed into a particular codeword c constituting an ECC code. In case any error is introduced in the communication channel, the received codeword r will be different from c. A conventional coding method for binary channels is $[n, k]$ linear block code, where

the message is split into consecutive blocks of k bits and each block is encoded into the codeword of n ($n > k$) bits.

It is shown in [54] that general decoding problem of linear codes is NP-complete. Particular classes of linear codes are linear-time encodable and decodable [181] which fits for the error correction in micro-architecture. Among such codes, Hamming code [148] is widely utilized due to its simplicity which gives less pressure to the critical path of the architecture. Usually, decoding of Hamming code can be executed in a single cycle.

For each integer r, there exists a Hamming code with codeword length $n = 2^r - 1$ and message length $k = 2^r - r - 1$. A generator matrix G with size $k \times n$ converts the message u into the codeword v by $v = uG$. Note that $r = n - k$ denotes the number of parity bits when G is written in the 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 the *extended Hamming code* which can detect double bit error but still correct single bit error. For 32 bits message, $r = 6$ parity bits are required to correct a single bit error, leading to 38-bit codewords. This scheme is denoted by $H_1[38, 32]$.

6.2.1.2 Divide and Conquer Method for Higher Bit-Error Correction

For higher bit error correction, one approach is to use BCH codes [143] or LDPC/turbo codes [147]. However, implementations for such codes are usually too slow to use in the instruction pipeline.

The total number of possible errors in a 32-bit message is $2^{32} - 1$. Out of this, the conventional Hamming code $H_1[38, 32]$ covers $\binom{32}{1} = 32$ cases. To tackle higher bit errors, the method of dividing the message into multiple segments and apply Hamming encoding and decoding on each segment in parallel is proposed. For instance, when a 32-bit word is divided into two halves, each of 16 bits have $r = 5$ parity bits on each half, therefore $(\binom{16}{0} + \binom{16}{1})^2 - 1 = 288$ cases can be covered. This scheme is denoted by $H_2[42, 32]$. Similarly, the 32-bit word can be divided into four parts of 8-bits each and apply a Hamming code with $r = 4$ parity bits on each part to achieve partial four-bit error-correction. This scheme is referred as $H_4[48, 32]$ and covers $(\binom{8}{0} + \binom{8}{1})^4 - 1 = 1295$ cases. Although 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 [205] Copyright ©2014 **IEEE**

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	$H_{32}[96, 32]$

Compared with conventional coding schemes in the communication system, the proposed scheme is properly designed for the instruction and data in processor storages with following reasons.

- Increasing block size for linear code tends to increase coding efficiency, therefore, reduces the codeword overheads. One downside of such approach is the significant increment in decoding and error correction time. By splitting data into different segments, the error correction can be performed in parallel without increasing the timing overhead.
- The proposed DCH code works perfectly on data in the processor pipeline, which are transmitted as a whole bundle of data bits (word). However, such approach cannot be used in the communication system. The reason is that communication systems involve complicated operations such as real-time framing and synchronization. The incoming bit streams are serially decoded which cannot be waited to group together before correcting the errors.

6.2.2 Exploration of Asymmetric Reliability

In this section, the DCH coding schemes are applied for asymmetric error protection on embedded processors including RISC and VLIW architectures. Both message-wise and execution-wise unequal protection techniques are demonstrated with different impact on hardware and application level reliabilities. Detailed fault injection experiments are investigated as well as the physical overheads caused by protection logic.

The LT_RISC and LT_VLIW processors from IPs of Synopsys Processor Designer are customized to incorporate asymmetric redundancy. Both processors are fully synthesizable under Faraday 90 nm CMOS technology and supported by CoSy compiler infrastructure [188]. High-level fault injection technique and EMR metric in Sect. 4.1 are used for reliability evaluation.

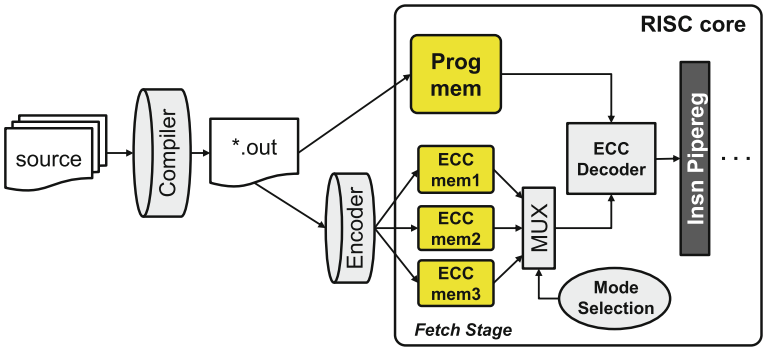


Fig. 6.12 Asymmetric protection for instructions on RISC processor [205] Copyright ©2014 **IEEE**

6.2.2.1 Impact of DCH Coding Levels

Figure 6.12 presents the flow of protection for the instructions on RISC processor. The compiled binary files are parsed into the encoder which generates different levels of Hamming codes according to Table 6.4. Small banks of ECC memories are introduced in the RISC core for loading the corresponding hamming codes according to their levels, while the regular instructions are loaded into the conventional program memory. During execution, the mode selection unit dynamically chooses the hamming codes from different ECC memories. The ECC decoder supporting different levels of ECC correction are used to detect and correct errors in the instructions. The instructions after ECC logic are fetched into the processor pipeline.

The DCH codes encoded by $H_1[38, 32]$, $H_2[42, 32]$, and $H_4[48, 3]$ support a maximum of 1, 2, and 4 bit error correction respectively. The effectiveness of different DCH modes is demonstrated by fault injection experiments, where bit-flips are randomly injected into both instruction and ECC memories. Figure 6.13 shows an example of amplified errors with increased number of faults. Each evaluation point is averaged from 1,000 experiments. The benchmark *sieve of Eratosthenes* is running on the RISC core. Huge gap of EMR is observed between the unprotected mode and protected ones. Among all DCH modes, $H_4[48, 3]$ achieves lowest EMR value which implies its strongest resistance to random bit-flip faults.

Table 6.5 presents the physical overhead for the RISC processor with different levels of protection under the estimation from Synopsys Design Compiler. As expected, the size of ECC memories increases for higher protection level. Interestingly, the area and power overhead are very close among all protection levels. The implementation under DCH $H_4[48, 32]$ achieves even smaller area than $H_1[38, 32]$ and $H_2[42, 32]$. The reason is that $H_1[38, 32]$ requires a larger encoder and decoder than other modes due to its arithmetic operation on 32-bit data as a whole. In contrast, the $H_4[48, 32]$ protects four separate data segment with the smallest circuit on each of them.

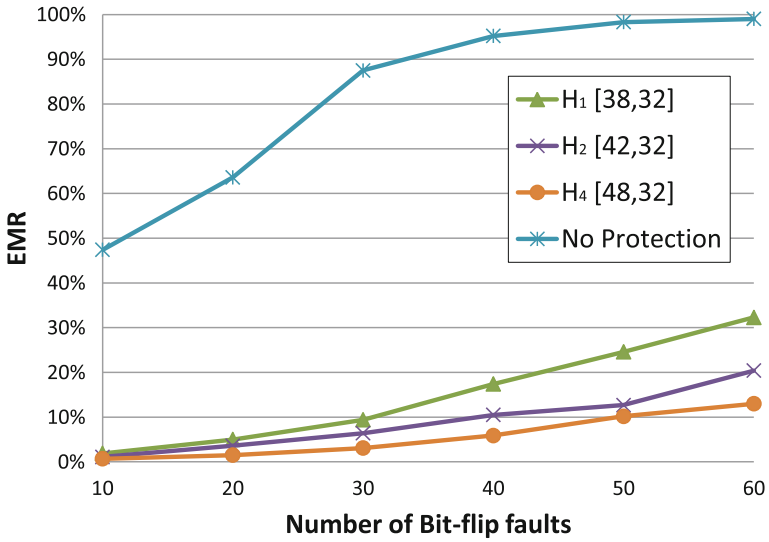


Fig. 6.13 EMR with different protection modes (Sieve application on RISC processor) [205] Copyright ©2014 **IEEE**

Table 6.5 Performances for different protection modes [205] Copyright ©2014 **IEEE**

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

6.2.2.2 Static and Dynamic Protection for Instructions

Different levels of DCH codes assist the processor to adaptively protect instructions. The instruction-wise criticality can be assigned statically or dynamically. The static approach is based on the observation that different types of instructions have different levels of impact to the results of the application. To characterize the impact of instructions, the same amount of bit-flip faults are injected into various types of instructions during the execution of benchmark applications. The corresponding EMR values are collected to compare the impact of errors among instructions. The Instruction Vulnerability Factor (IVF) is used to characterize the criticality of different instructions. IVF is defined as the EMR caused by faults on single types of instructions. Figure 6.14 shows the IVF for instructions in the RISC processor. According to the IVF distribution, the instructions are statically classified into three criticality levels.

On the other hand, the dynamic approach switches instruction-wise protection levels according to the runtime condition of error detections. The idea is that more

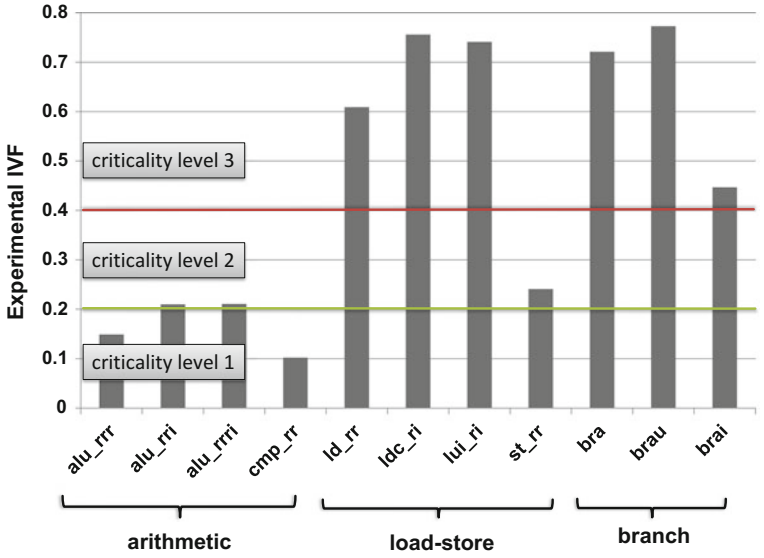


Fig. 6.14 Static instruction criticality assignment [205] Copyright ©2014 **IEEE**

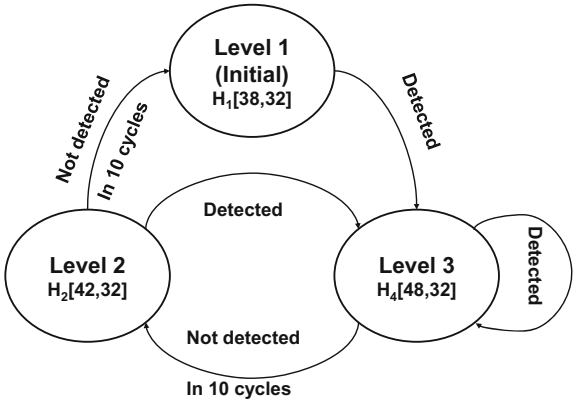


Fig. 6.15 FSM for dynamic, asymmetric reliability [205] Copyright ©2014 **IEEE**

detected errors switch the processor to higher protection levels while fewer errors tend to reduce levels with less power consumption. Figure 6.15 shows the state transition diagram as the FSM of the level controller. Initially, level one is assigned. In case one error is detected, the processor is switched to level three for a fixed length of clock cycles (10 cycles for instance). If no errors are detected in this period, the level controller gradually reduces the protection level, until it reaches and keeps at level 1.

The static approach requires additional coding of instruction levels based on Fig. 6.14 and the mode selection unit in Fig. 6.12 parses the additional coding fields to give signals to the multiplexer. For the dynamic approach, the mode selector is enhanced with the level control FSM, where the error condition is provided by the

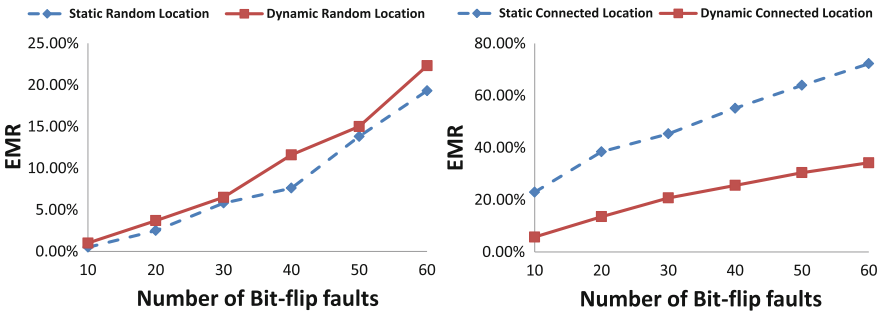


Fig. 6.16 Comparing static and dynamic protection [205] Copyright ©2014 **IEEE**

ECC decoder. Static and dynamic modes can be switched conveniently through a special flag register.

Fault injection is performed to compare the protection efficiency under both approaches of instruction-wise asymmetry redundancy. Two schemes of experiments are designed. The first scheme models ground-level errors by injecting bit-flips into random locations of the instruction memory. The second scheme injects multiple word errors in adjacent memory locations to imitate the impacts under radiation of strong particles. Shown in Fig. 6.16, the static approach gives slightly better error resilience under the errors at random memory location. However, the dynamic approach achieves significantly lower EMR for adjacent errors. Especially, for errors on instructions within a loop the dynamic approach has much higher error coverage since the rising of protection level maximally protect all instructions in the loop regardless of the types of instructions.

6.2.2.3 Asymmetric Reliability for VLIW Processor

As discussed in Sect. 6.1.2.2, VLIW architectures provide extra spatial redundancy due to the idle instruction slots. This is usually caused by the limitation of parallelism in the program for the compiler to explore. The idle slots containing NOP instructions can be used to store the encoding of various modes of DCH code instead. Since such packaged DCH codes flow through instruction pipeline, they can be used to synchronously decode and correct instructions in the meaningful slots. Optimally, two VLIW syllables are filled with meaningful instructions while the rest two are filled with DCH codes to correct corresponding instructions. Figure 6.17 visualizes the scheme where ECC denotes the pipeline register containing the DCH codes.

The instruction slot of 32 bits can be ideally filled with all three modes of DCH parity bits. However, one bit is required to identify whether the slot contains instruction or ECC, while another bit is to identify the end of the instruction. As a result, only two DCH modes $H_1[38, 32]$, and $H_4[48, 32]$ are filled into the ECC slots. The static and dynamic approaches of level assignment select the protection levels during the execution.

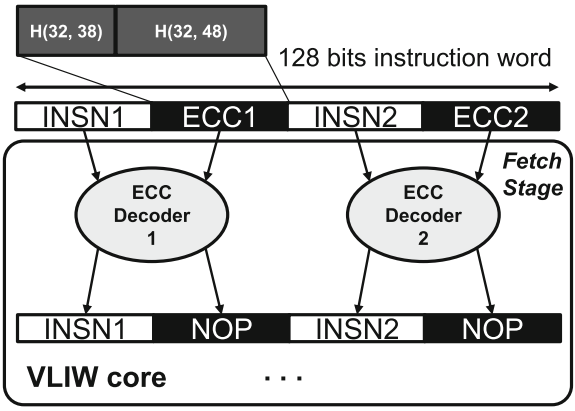


Fig. 6.17 ECC in VLIW slots [205] Copyright ©2014 **IEEE**

Table 6.6 Reliability versus power/area trade-off [205] Copyright ©2014 **IEEE**

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

Table 6.6 shows the physical overheads of the protection where both area and power increase with the hardware support of increased protection levels. Another design metric is the application time. Since the original assembly program does not always leave two empty instruction slots, decisions can be made to whether compulsorily protect all instructions or opportunistically protect instructions with available slots. The compulsory approach forces the pipeline to be always filled with two valid instructions, therefore increases the execution time of any application. The opportunistic approach does not achieve performance penalty but has less instruction coverage. To investigate the trade-off between reliability and application time, fault injections are performed to compare the EMR under both modes. For simplicity, the DCH $H_4[48, 32]$ code is used to protect the instructions.

Figure 6.18 shows the trends of EMR with a number of injected errors in the program memory. For all applications, compulsory protection achieves lower EMR than the opportunistic one. However, their EMR gaps vary among applications. Table 6.7 indicates the performance overhead of the compulsory mode. It is interesting that larger performance penalty also results in smaller EMR values. Such trend implies the potential in the asymmetric design methodology joint considering software parallelism, execution time and reliability.

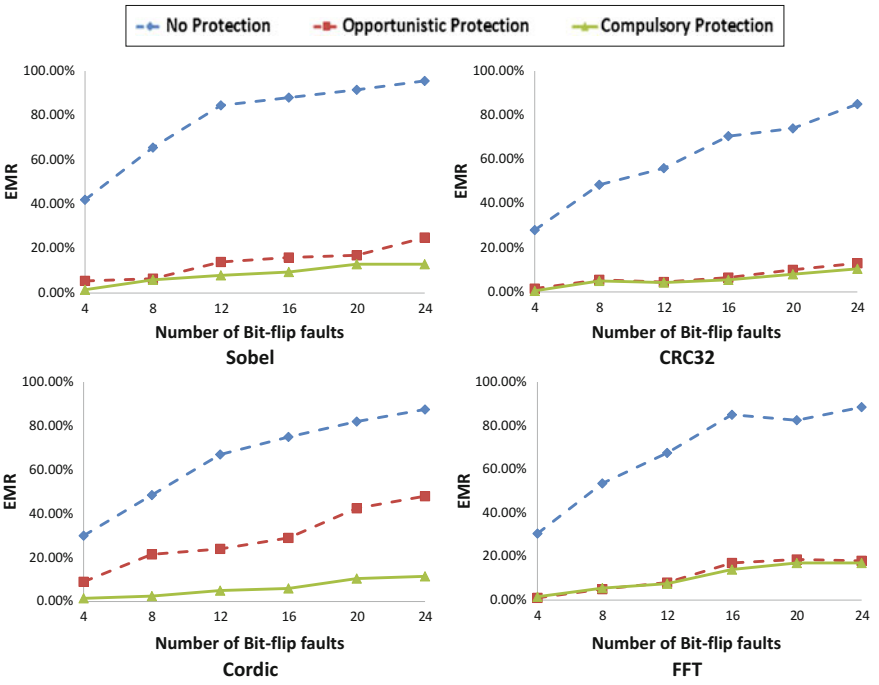


Fig. 6.18 EMR for different VLIW protection modes [205] Copyright ©2014 **IEEE**

Table 6.7 Application runtime for different VLIW protection modes [205] Copyright ©2014 **IEEE**

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

6.2.2.4 Bit-Wise Asymmetric Reliability for Data Memory

In contrast to the message-wise asymmetric encoding for the instructions, data memories are prone to be protected through bit-level asymmetry. This is based on the observation that MSBs in the data message lead to higher significance to the computational results than the LSBs. Consequently, MSBs can be encoded with higher level DCH codes than the LSBs. Figure 6.19 illustrates an example for asymmetric encoding for data words, where the eight MSBs are encoded into two segments of DCH $H[7, 4]$ and the rest bits into three segments of DCH $H[12, 8]$. The DCH

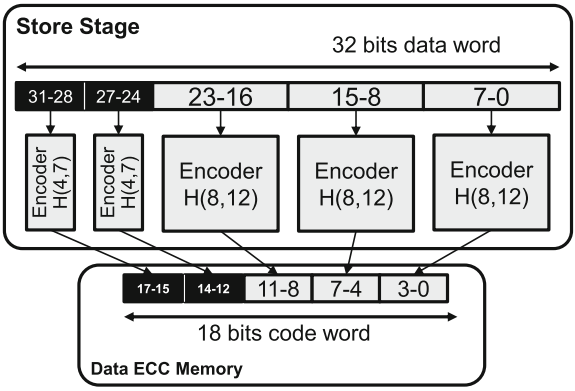


Fig. 6.19 Bit-wise asymmetric encoding [205] Copyright ©2014 **IEEE**

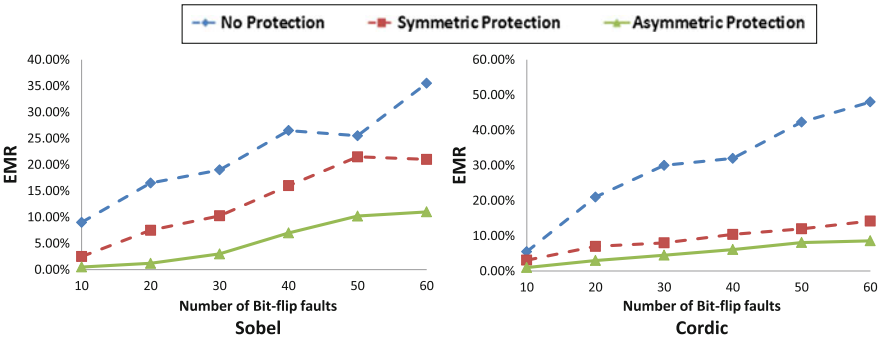


Fig. 6.20 Comparing symmetric and asymmetric bit-wise protection [205] Copyright ©2014 **IEEE**

codes can be encoded and stored into separate ECC memory during conventional store instruction, while retrieved for error correction during load instruction.

The proposed asymmetric protection is benchmarked with symmetric encoding where four bytes of 32 bits data are encoded into four DCH $H[12, 8]$ codes. Errors are randomly injected into the data memory for two applications. Figure 6.20 presents the trends of EMR which show that asymmetric protection for data achieves lower error rates than the symmetric one for both applications. It is also observed that the code size of asymmetric mode (18) is only 2 bits more than the size for symmetric mode (16). Such advantage is also application dependent, since the programs, which never involve the calculation using MSBs, will not show any effects in the scale of errors with asymmetric encoding.

6.2.3 Summary

This work introduces various perspectives of asymmetric reliability for processor design, which include the static/dynamic, opportunistic/compulsory, and message-wise/bit-wise asymmetry. The DCH coding schemes are proposed to unequally protect the individual messages. Implementations on RISC and VLIW processor demonstrate the design methodology with detailed fault injection experiments for error evaluation.

6.3 Statistical Error Confinement

The aggressive shrinking of transistors has made circuits and especially memory cells more prone to parametric variations and soft errors that are expected to double for every technology generation [23], 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 [169], thus indicating the need for immediate adoption of effective fault tolerant techniques.

Techniques such as Error Correcting Codes (ECC) [57] and Checkpointing [52] 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 [32] 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 [110, 214]. Few very recent approaches exist also that extend generic instruction sets with approximation features and specialized hardware units [60, 166, 194]. 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 an 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 an 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 sta-

tistical characteristics, while allowing to mitigate single and multiple bit errors with substantially fewer overheads.

6.3.1 Proposed Error Confinement Method

Assume that a set of data $d \in \mathcal{D} = \{d_1, \ldots, d_K\}$ 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

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

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 the case of apriori 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:

$$d^* = g(\mathcal{F}) \triangleq \underset{\bar{d}}{\operatorname{argmin}}\ \mathcal{C}(\bar{d} \mid \mathcal{F}). \tag{6.2}$$

where,

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

Basic arithmetic manipulations show that the resulting correction function is given by $g_{\mathrm{MMSE}} = \mathbb{E}\{d[n] \mid \mathcal{F}\}$. 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 an RISC Processor

The proposed Error-Confinement function requires a scheme for detecting a memory error in order to provide the needed apriori information $\mathcal{F}$ and a look-up table for

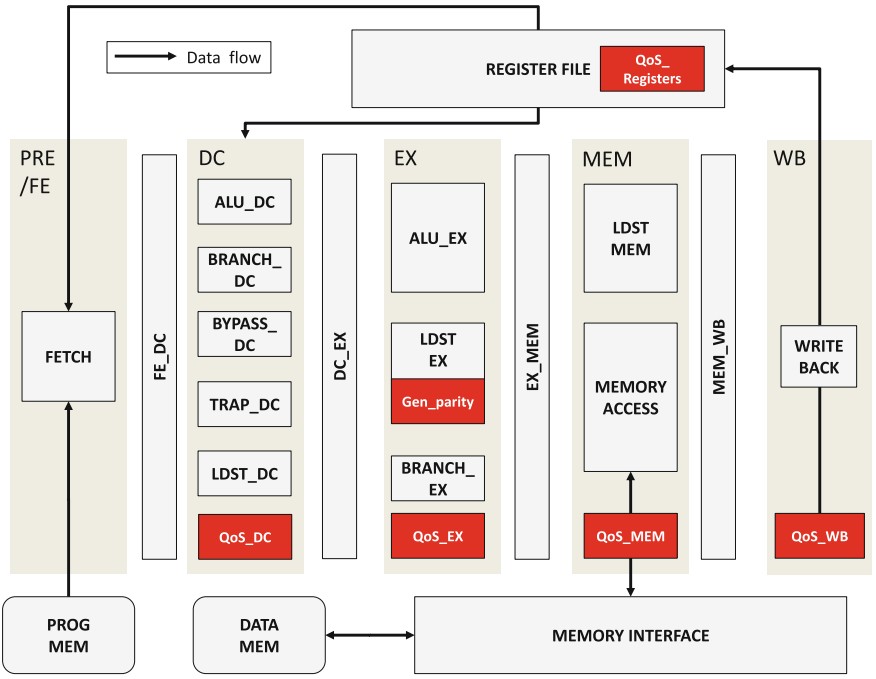


Fig. 6.21 Microarchitecture of RISC processor with enhancements for statistical based error confinement [202] Copyright ©2016 **IEEE**

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 [184], which consists of five pipeline stages as depicted in Fig. 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 the 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.

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 [184]), 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 Fig. 6.21. The detailed functionality of the logic functions within each module in each pipeline stage is described in detail in Fig. 6.22.

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 a 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 Fig. 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) [70] 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 ratio (PSNR) [87] and a typical PSNR value for a lossy image is 30 dB.

QoS_DC

DC stage module to decode 4 new instructions in ISA:

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

EX stage module, implement following operation:

1. ***parity*** in MEM stage is activated in case the stored data lies in the protected range

QoS_EX

EX stage module, implement following operation:

1. ***chk_ld_ex***: Activated by ***chk_ld***, verify parity bit of the incoming data. In case of missmatch, activate ***chk_ld_mem*** in MEM, otherwise activate ***chk_ld_wb_correct***

QoS_ MEM

MEM stage module, implement following operations:

1. ***parity***: Main parity encoding logic. Each 32 bits data generates 1 bit parity. Encode 32 words as 1 word parity, activate ***partiy_st*** logic in WB stage 32 bits parity are ready
2. ***chk_ld_mem***: When a parity missmatch is detected, get the address of corresponding LUT element and issue memory read. Activate ***chk_ld_wb_wrong*** in WB stage

QoS_WB

WB stage module, implement following operations:

1. ***parity_st***: Store 32 bits parity as one data word into data memory
2. ***Chk_ld_wb_wrong***: In case of parity missmatch, return the reference value from LUT
3. ***Chk_ld_wb_correct***: No parity missmatch, return the input value from instruction

QoS_ Registers

Special registers in RegisterFile,

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.

Fig. 6.22 Introduced modules and their functionalities [202] Copyright ©2016 **IEEE**

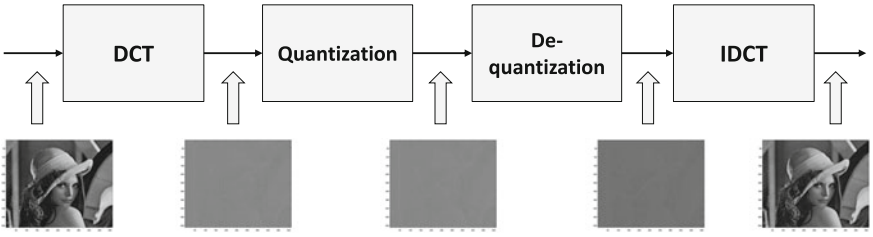


Fig. 6.23 Subsystem in JPEG application [202] Copyright ©2016 **IEEE**

6.3.3.2 Statistical Analysis of JPEG

Following the steps of the proposed approach, the different stages of JPEG are statistically analyzed 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 an LUT that was described in Sect. 6.3.2.

-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

(a) Average 8x8 matrix after DCT operation

-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

(b) Average 8x8 matrix after Quantization

Fig. 6.24 Reference matrix for DCT and quantization coefficients [202] Copyright ©2016 **IEEE**

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 the inline assembly to describe JPEG as shown in Fig. 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.

```
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);
    ......
}
```

Fig. 6.25 Programming example with custom instructions for DCT [202] Copyright ©2016 **IEEE**

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 65 nm process technology. The results show that in the 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.

Fig. 6.26 Output images under different schemes of error injection [202] Copyright ©2016 **IEEE**

As mentioned the quality achieved by the proposed approach is compared with an SECDED ECC. Figure 6.27 shows the obtained results in the case of protecting the output of DCT and quantization coefficients with the two schemes under a 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 the 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

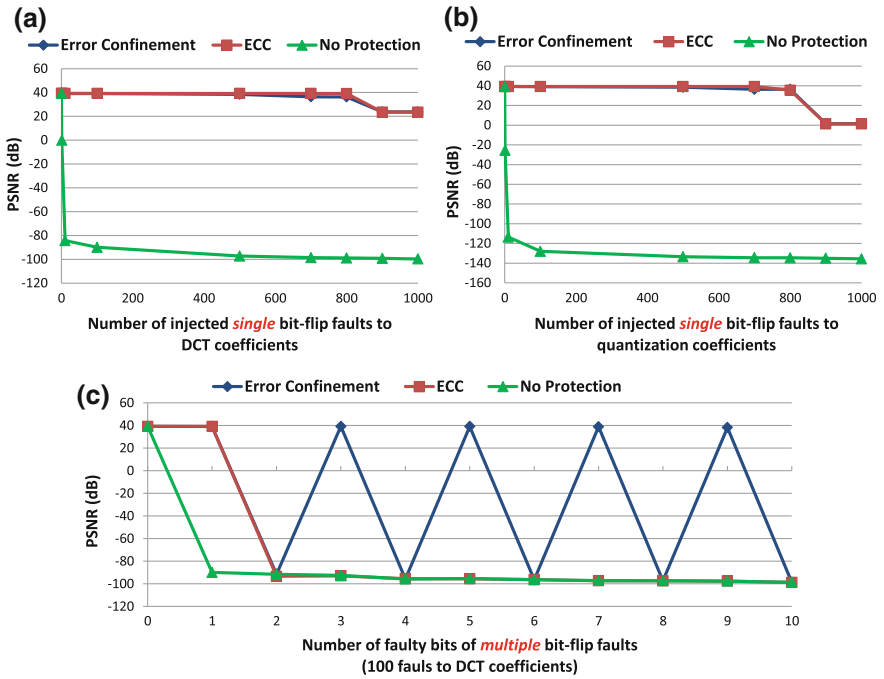


Fig. 6.27 PSNR under no protection, proposed scheme and ECC [202] Copyright ©2016 **IEEE**

cannot detect two bit-flips in a word and thus it does not engage the replacement of the erroneous data.

The results reveal also a different aspect in the JPEG application. In particular, it is observed that in the 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 a different number of bit-flips in each word. It is observed that the proposed scheme helps to obtain a PSNR of more than 38 dB (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 fails 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

Table 6.8 Results for the proposed architecture extensions compared to the reference unprotected processor [202] Copyright ©2016 **IEEE**

	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

multi bit-flips in a single word. The addition of another parity could be employed to improve the capability of 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 65 nm 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 the 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 an H (38, 32) ECC will result in much larger overheads.

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 Fig. 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.

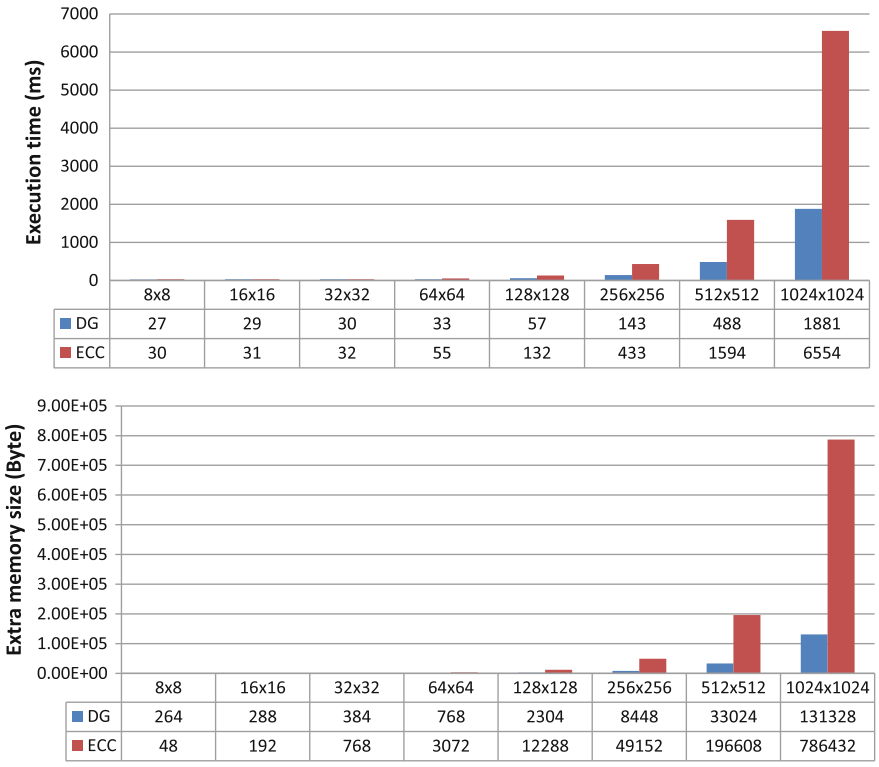


Fig. 6.28 Execution time, data memory usage for error confinement versus ECC [202] Copyright ©2016 **IEEE**

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 Fig. 6.29. This is because ECC takes a 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 Fig. 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 an RISC processor with

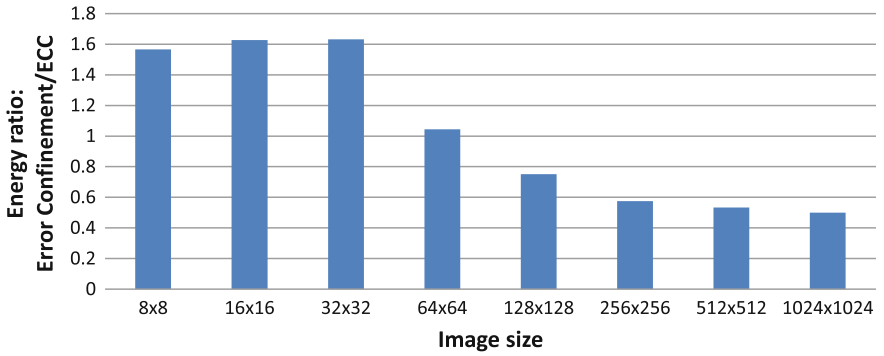


Fig. 6.29 Energy ratio between error confinement and ECC versus image size [202] Copyright ©2016 **IEEE**

custom instructions supporting proposed approach is presented. The benchmarking 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

System-level reliability focuses on the problem of executing specific applications correctly on multi/many core systems. The literature on such issues can be classified in two directions: reliable task mapping and reliable network design. In this chapter two techniques to enhance reliability in system-level design are proposed. First, a system-level exploration framework is presented in Sect. 7.1 which supports the 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 Sect. 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 modeling 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 a 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 [174]. With regard to the techniques improving device lifetime, [46] discusses approaches for addressing the lifetime optimization in terms of Mean-Time-To-Failure (MTTF). Coskun et al. [41] presented a temperature-aware mapping that leads to increased lifetime. A wear-based heuristic is proposed in [80] to improve the system lifetime.

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_7

On the other hand, several papers target reliable mapping in presence of transient faults. In [109] the authors propose a remapping technique aimed towards determining task migrations with the minimum cost while minimizing the throughput degradation. In [168] a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems is presented. [49] 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 [74] 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 [96] streaming applications is introduced. The processing elements and communication channels are equipped with fault injection properties proposed in Sect. 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 the software developer. For instance, higher reliability levels can be assigned to node P3 and P6 in Fig. 7.1 due to higher degree of edges. From the architecture side, the ability to integrate customized processor helps improving core-level reliability.

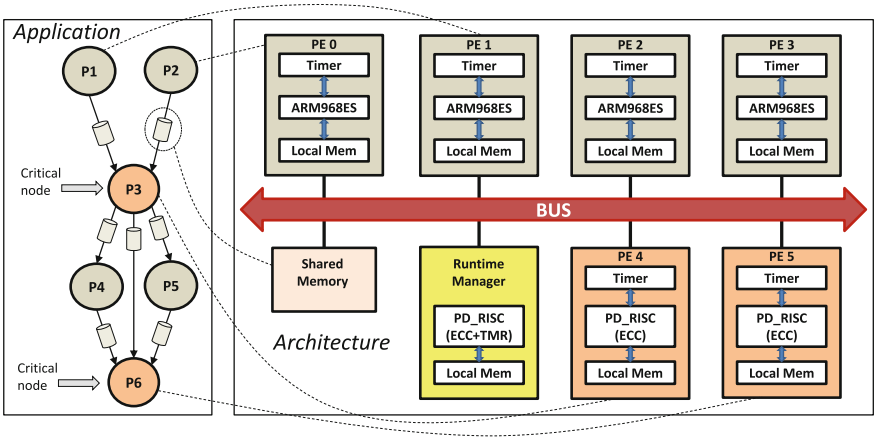


Fig. 7.1 KPN tasks mapping to MPSoC considering node reliability level [201] Copyright ©2014 **ACM**

In Fig. 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, the task with high-reliability level is 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. The timer on individual processor informs the manager core whether the monitored processor is in an 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 [184], customized processor in both RTL and SystemC package can be automatically generated from high-level descriptions. 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 Sect. 4.1 is applied for the 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 Fig. 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, the 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 neighboring PE nodes and connecting channels. For instance, bus based platform in Fig. 7.1 is configured as a processor network with the full connection. Architectural reliability index for each core is defined according to the EMR metric in Sect. 4.1.1.4. Detailed EMR evaluation for heterogeneous processors is referred in [203]. 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 The platform, channels are implemented as data structures in shared memory according to the topology to emulate generic styles of interconnects, which gets automatically analyzed from topology graph. Fault injection in channels

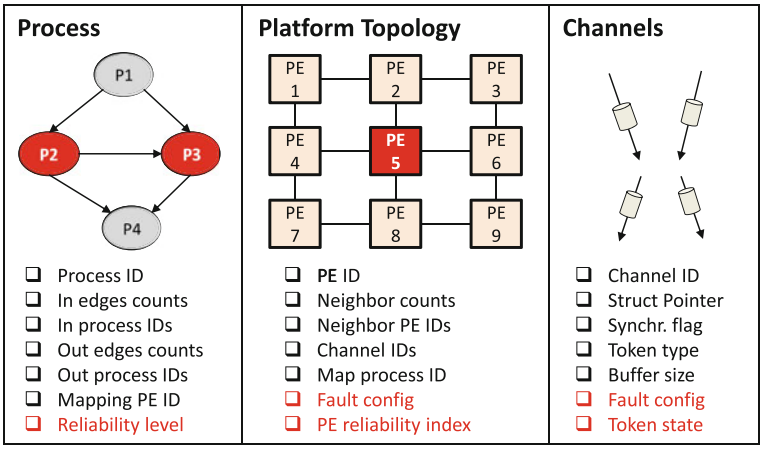


Fig. 7.2 Data structures for platform initialization [201] Copyright ©2014 **ACM**

is implemented as bit manipulation in the data elements of the 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 Fig. 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 to retrieve the processing state when run-time task remapping happens. After remapping, the start node can directly process the next token (Fig. 7.3).

State transition Upon system initialization, the manager initializes KPN processes, topology, and channels according to user-provided information while PEs wait for task 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 the case of mapping failure, the run-time manager terminates the system. A successful mapping will interrupt all PEs for task switching while the 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 [174].

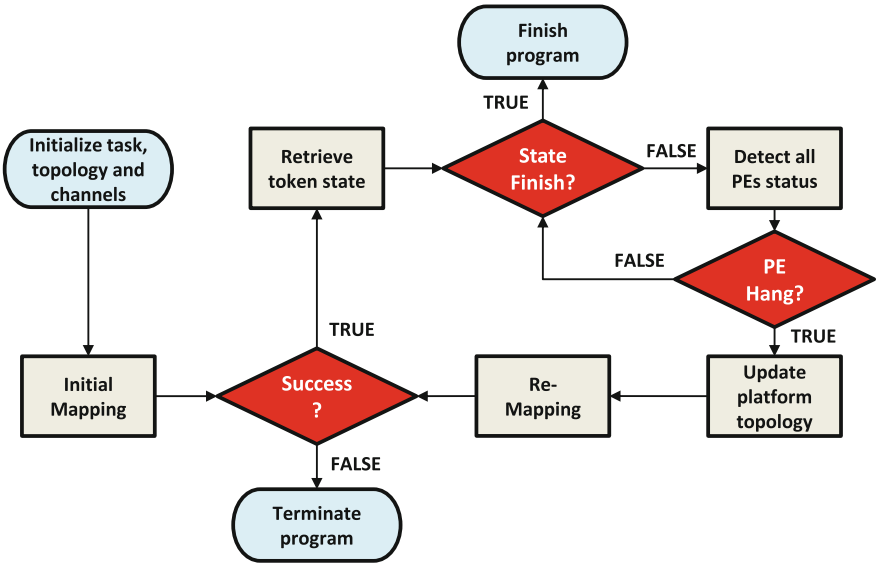


Fig. 7.3 Run-time manager state transition [201] Copyright ©2014 **ACM**

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 [201] Copyright ©2014 **ACM**

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)                    ▷ init recursive mapping
5:     return status
6: end function
7:
8: function RECURSIVEMAP(task_id)
9:     if task_id == task_Count then
10:        return Success                          ▷ last task has been mapped
11:    end if
12:    for pe_id = 1 to PE_Count do
13:        if mapT2P(task_id, pe_id) then          ▷ mapping plug-in
14:            binding(PE[pe_id], TA[task_id])
15:            if RECURSIVEMAP(task_id + 1) == Success then
16:                return Success                  ▷ recursive mapping success
17:            else
18:                PE(pe_id) → t_id = null
19:            end if                              ▷ recursion fail, clear parent decision
20:        end if
21:    end for
22:    return Fail
23: end function
```

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 [201] Copyright ©2014 ACM

```
1: function MAPT2P(task_id, pe_id)
2:     if PE(pe_id) → Degree < TA(task_id) → Degree then
3:         return Fail                                   ▷ meet task edges constraint
4:     end if
5:     if PE(pe_id) → relia_ind < TA(task_id) → relia_lev then
6:         return Fail                                   ▷ meet reliability requirement
7:     end if
8:     neighbors_ids = get_task_neighbors(task_id)
9:     for all neighbors_ids do
10:        pe_neb_id = TA(neighbors_ids) → 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                  ▷ ensure task neighbors are topological neighbors
17:    end for
18:    return Success
19: end function
```

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 neighbors. 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 neighbors.

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 Fig. 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

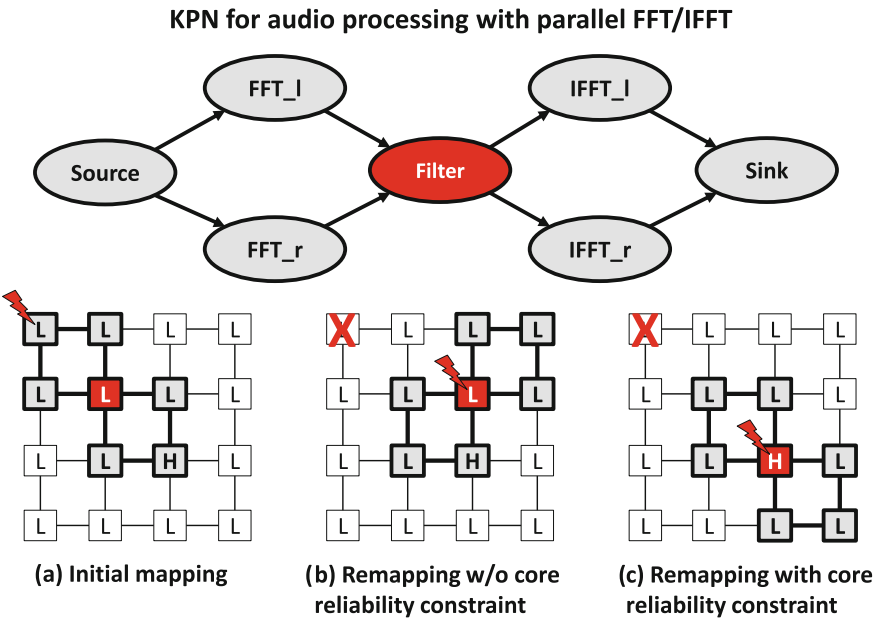


Fig. 7.4 KPN tasks mapping onto 16 PE platform [201] Copyright ©2014 **ACM**

proposed mapping algorithm, the platform consists PD_RISC processors with ECC protection on its program counter register (PC-register), which is labeled as 'H' while the rest ARM processors are labeled as 'L'.

7.1.3.1 Algorithm Constraints

Initially a fixed mapping as in Fig. 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 the unresponsive state which activates the run-time manager for task remapping. When only edge count constraint is applied, the tasks are mapped as in Fig. 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 Fig. 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.

Table 7.1 Mapping exploration with different algorithm constraints [201] Copyright ©2014 **ACM**

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

7.1.3.2 Topology and PE Types

Further mapping explorations with different topology and PE types are performed as in Fig. 7.5. A platform with mesh topology suffers from 3 unresponsive PEs as in Fig. 7.5e. One extra high reliable core does not facilitate further remapping as shown in Fig. 7.5f. In the contrary, a topology with more links such as nearest neighbor (NN) realizes further mappings, where up to 5 unresponsive PEs are tolerant as in Fig. 7.5i. When further highly reliable core is deployed, remapping is still achieved with 6 hanging PEs as shown in Fig. 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 Fig. 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-kilocycles on the supervisor core for 16-PE Mesh topology and 138-kilocycles 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 the only difference of a few cycles is incurred during PE initialization.

The approach of design time analysis [168] is adopted and keep the mapping results in the local memory of task manager so that mapping decisions can be directly 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

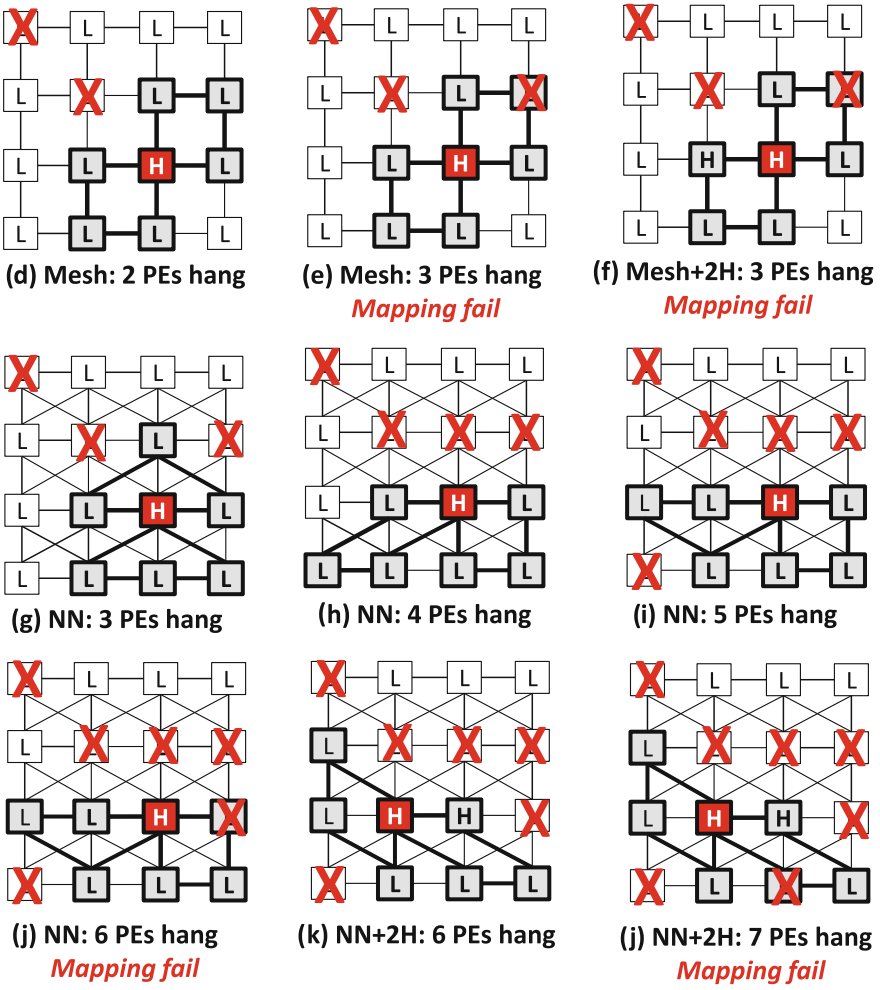


Fig. 7.5 Mapping exploration for 7 KPN nodes [201] Copyright ©2014 **ACM**

user-defined architecture topology and its ability to integrate customized processors with reliability extension demonstrate the usability of the proposed mapping technique.

Table 7.2 Exploration with topology and PE types [201] Copyright ©2014 **ACM**

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	–

7.2 Reliable System-Level Design Using Node Fault Tolerance

In parallel to task mapping techniques [174], construction of reliable network topology is another research direction to increase system-level reliability. In the domain of Network-on-Chip (NoC) design, various NoC topologies have been investigated in order to minimize routing delay in presence of failure nodes/edges. For instance, Mesh [193], Torus and Tree [179] are popular topologies in both academia and industry. Customized topologies such as de Bruijn graph [82] incurs less latency overhead and energy consumption than Mesh and Torus facing faulty edges. For detailed evaluation on reliable network topologies, readers are referred to the survey paper in [63]. Despite the generality of usage among different NoC applications, such popular topology is usually over-designed for customized tasks which incur a lot of unnecessary edges. The ad-hoc reliable network topology is desirable for customized tasks.

The Kahn Process Networks (KPN) [69] is a general network model to describe applications involving multiple processes. In the KPN model, streams of data propagate through the processing nodes which are connected in serial or parallel. Each node in KPN can be viewed as one processing element in the many-core system, whereas the edges can be treated as the on-chip storages (FIFOs, RAMs). Since KPN is a standardized model for signal processing applications, the methodology to design reliable network topology for a given KPN is of high importance. Formulated from the perspective of graph theory as the problem of Node Fault Tolerance (NFT), the goal is to construct a supergraph which is isomorphic to the given KPN graph when any of its nodes and connecting edges is removed. Furthermore, finding such NFT supergraph with the smallest amount of edges is the problem of constructing optimal NFT. Previous literature [79] only handles the construction of optimal NFT graph for a subset of simple graphs such as path and circle. However, finding the optimal or near-optimal NFT for a generic task graph in KPN model is a very challenging topic.

Contribution In this work, a divide-and-conquer based methodology to construct NFT graphs for generic KPN task graph is introduced. The generic graph is first decomposed into a set of subgraphs which can be individually handled by the theory of optimal NFT in [79]. After that, the individual optimal NFT graphs are merged together to form a supergraph which is the NFT graph of the original task graph. Further edge reduction can be performed on the supergraph using exhaustive search based algorithm. It is shown that the proposed graph construction algorithm achieves optimal 1-NFT graph even without edge reduction. The proposed algorithm is demonstrated through failure injection experiments by the system-level exploration framework in Sect. 7.1.

7.2.1 Node Fault Tolerance in Graph

This section provides the background information on the graph theoretical models by Harary and Hayes [79]. Two graphs involved in the theory of NFT are G and G*. Graph G is always embedded in G* (graph isomorphic) when a certain amount of nodes and their adjacent edges are removed from G*. G* is termed as *k-NFT(G)* when the removal of maximal k nodes is tolerated in G*. There are many G* fulfilling the requirements of k-NFT. Out of those graphs, the G* with the smallest amount of edges is termed as the *optimal k-NFT(G)*. Note that the optimal k-NFT(G) graphs are also not unique for graph G. In conclusion, the procedure of finding the optimal k-NFT(G) can be stated as following:

- Construct a set of supergraph G* whose members are all k-NFT(G).
- Find the optimal k-NFT(G) among the set of G*.

For instance, a circle G with 5 nodes is denoted as C_5 in Fig. 7.6a. A graph variant of 1-NFT(C_5) with 10 edges is present in Fig. 7.6b, which introduces a spare node (s). When the node (f) is removed, C_5 is isomorphic to the supergraph G* with the spare node, which is highlighted as the bold circle. Figure 7.6c shows the optimal 1-NFT(C_5) with 9-edges. Besides, Fig. 7.6d gives the optimal 2-NFT(C_5) with 14-edges.

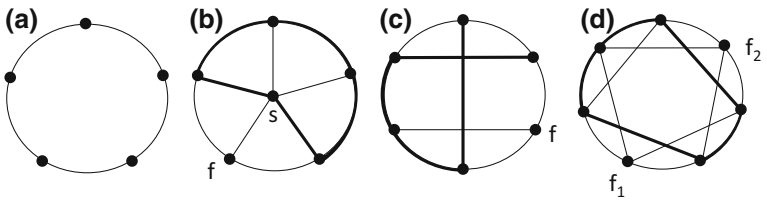


Fig. 7.6 **a** circle C_5; **b** non-optimal 1-NFT(C_5); **c** optimal 1-NFT(C_5); **d** optimal 2-NFT(C_5); [79] [204] Copyright ©2016 **IEEE**

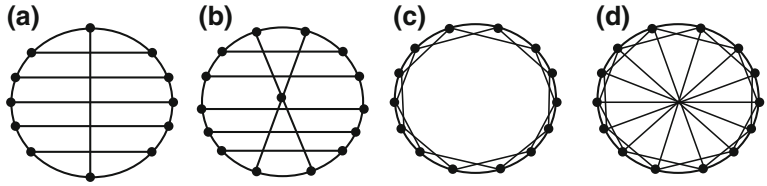


Fig. 7.7 Exemplary NFT graphs for **a** 1-NFT(C_n) n odd; **b** 1-NFT(C_n) n even; **c** k-NFT(C_n) k even; **d** k-NFT(C_n) k odd; [79, 204] Copyright ©2016 **IEEE**

7.2.1.1 Optimal Node Fault Tolerance

Harary and Hayes [79] presented the theory to construct optimal k-NFT graph for circles and paths, which are stated as:

Harary-Hayes Theorem 1: *The two Hamiltonian-connected (n+1)-node supergraphs in Fig. 7.7a and b show the optimal 1-NFT* (C_n) *for odd and even number of n.*

Let $k = 2h$ for even k and $k = 2h + 1$ for odd k. C_n^m indicates the power graph, which is obtained by connecting each node i in C_n to all of its nodes at distance m and less.

Harary-Hayes Theorem 2: *For even k, the power graph* C_{h+k}^{h+1} *in Fig. 7.7c is an optimal k-NFT* (C_n). *For odd k, the graph obtained by adding [(n+k+1)/2] bisector edges to* C_{h+k}^{h+1} *as Fig. 7.7d is an optimal k-NFT* (C_n).

Harary-Hayes Theorem 3: *Let* P_n *be the 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 Theorems 1 and 2.

7.2.2 Construct NFT for Generic Graph

An algorithm which constructs the NFT graph for generic graphs is extended from the theorems of Harary and Hayes. The heuristic takes advantage of the *divide and conquer* technique to construct optimal NFT graph for elementary subgraphs supported by the existing theorems. Algorithm 4 illustrates the proposed divide and conquer approach to constructing the k-NFT 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 constructed by merging G_e^* with G^* in the current iteration. For simplicity, the procedure of labeling elementary graphs is not present in Algorithm 4.

Algorithm 4 Constructing k-NFT(G) for arbitrary graph G [204] Copyright ©2016 **IEEE**

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 $G_e^* = k - NFT(G_e)$ ▷ according to Sect. 7.2.1
6: $G^* = merge(G_e^*, G^*)$
7: **end for**
8: **return** G^*
9: **end function**

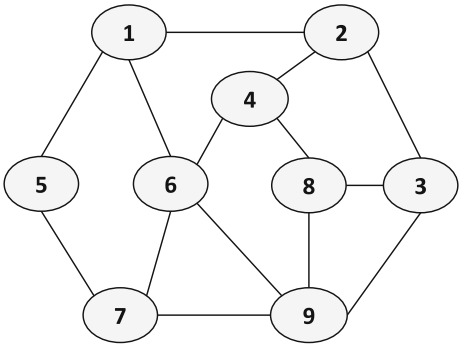


Fig. 7.8 The task graph G with nine nodes [204] Copyright ©2016 **IEEE**

The construction of G_e^* from G_e is guided by the theorems in Sect. 7.2.1. The procedure of graph merging is illustrated on a task graph with 9 nodes and 6 elementary circles in Fig. 7.8.

Figure 7.9 presents the C_3 and C_4 graphs with their 1-NFT and 2-NFT supergraphs which can be constructed using Harary-Hayes Theorems 1 and 2. The spare nodes are shown in black color. Since no other elementary graphs are identified in Fig. 7.8, other constructions of NFT graphs are not listed. To represent the graph, the concept of adjacency list [40] is adopted, where the neighbour nodes of each one is represented as a list such as the one shown in Fig. 7.10a. For instance, Fig. 7.10b present the 1-NFT(C_4) graph in Fig. 7.9e.

Figure 7.10 shows three adjacency list representations of 1-NFT graphs for elementary circles and the list merging procedure represented as the matrix operation. To merge two lists 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 elementary circles which is put at 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

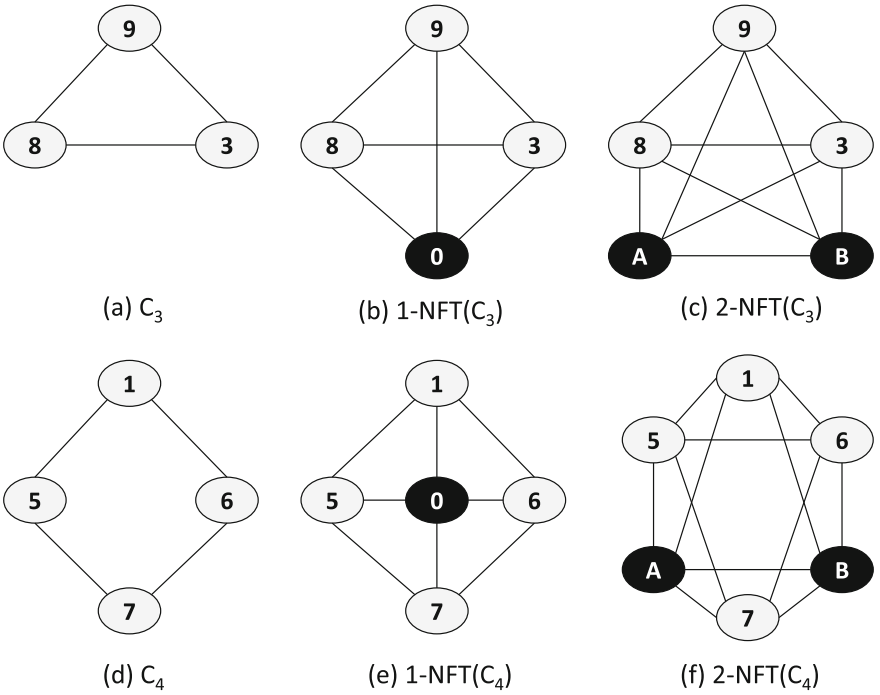


Fig. 7.9 Optimal 1-NFT and 2-NFT graphs for C_3 and C_4 [204] Copyright ©2016 **IEEE**

1-NFT and 2-NFT graphs for G in Fig. 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 Sect. 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

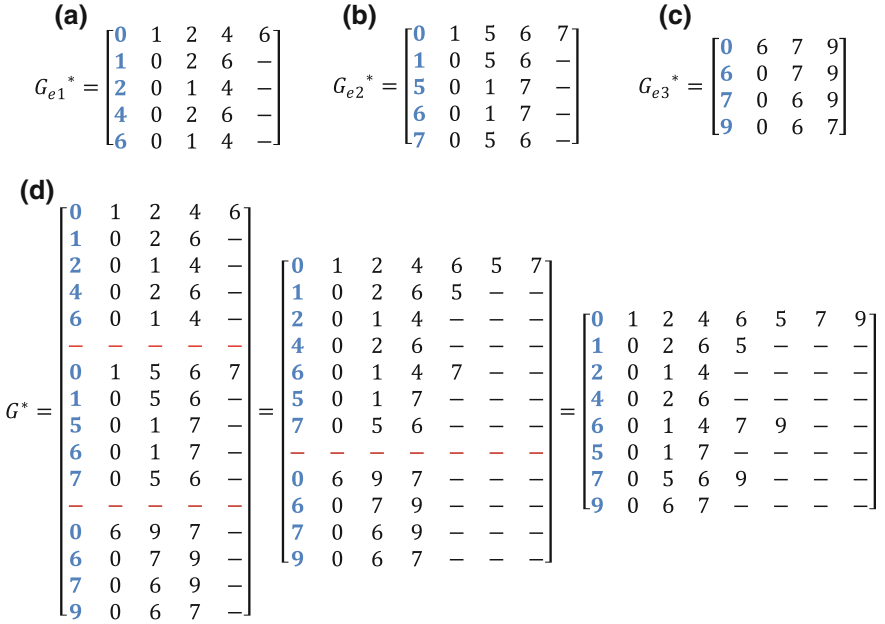


Fig. 7.10 Merge of three 1-NFT graphs [204] Copyright ©2016 **IEEE**

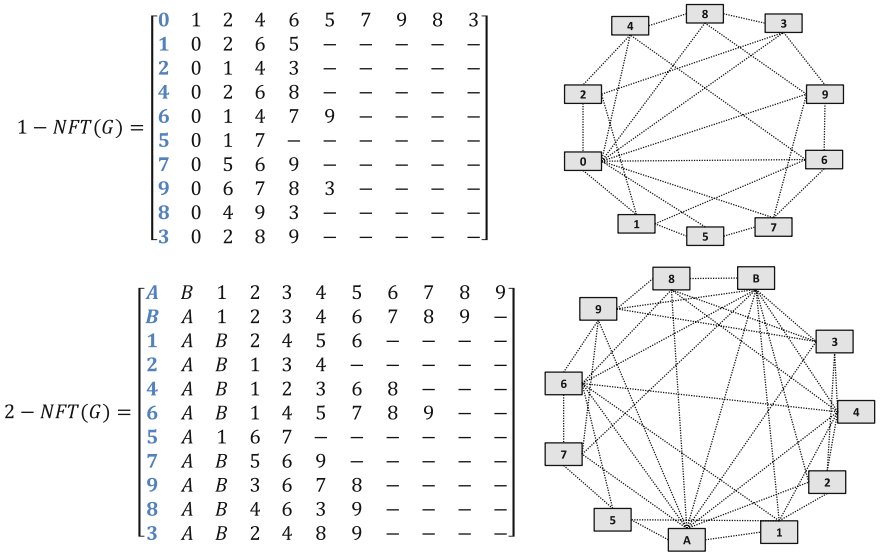


Fig. 7.11 Final 1-NFT(G) and 2-NFT(G) [204] Copyright ©2016 **IEEE**

which takes advantage of the recursive task mapping procedures from Algorithms 2 and 3 in Sect. 7.1.2.

Algorithm 5 Reduce number of edges in graph G [204] Copyright ©2016 **IEEE**

INPUTS: G: Task graph; G^*: Output task graph from Algorithm 4
OUTPUT: G^*_{opt}: Locally optimal k-NFT graph

```
1: function NFT_REDUCTION(G, G*)
2:     status = NFT_VERIFY(G, G*, k)
3:     if status == 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=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=RUNMAP(G_chk, G)                  ▷ Algorithms 2 and 3 in Sect. 7.1.2
21:        if status == Fail then return Fail
22:        end if
23:    end for
24:    return Success
25: end function
```

Algorithm 5 constitutes two procedures, the NFT_REDUCTION which reduces the number of edges and NFT_VERIFY which verifies the correctness of NFT for the given G and produced G^*.

- NFT_VERIFY: attempts to remove k nodes and their connecting edges from G^* and applies the task mapping procedure to map G onto the updated G^*. G^* is $k - NFT(G)$ when at least one possible mapping scenario is available under the removal of any selection of k nodes and adjacent edges in G^*.
- NFT_REDUCTION: After verifying that G^* is $k - NFT(G)$, this procedure attempts to remove one edge per iteration out of all edges and check if the resulted graph is still edged reducible. The recursive function ensures that G^*_{opt} is with the smallest number of edges based on the initial graph G^* (local but not globally optimal).

Algorithm 5 takes the input of the k-NFT graph G^* produced from Algorithm 4 which results in a locally optimal k-NFT graph. Besides, Algorithm 5 can as well

take the input of a fully connected graph G^* so that the globally optimal graph G^*_{opt} can be achieved through edge reduction. However, such exhaustive search from the full connection graph with 10 nodes can recursively make up to 2^{45} choices for edge removal in NFT_REDUCTION, which is exponential in timing complexity even for the 1-NFT graph. Consequently, providing G^* from the analytical approach reduces the maximal number of choices to 2^{23} since only 23 edges remain in $1 - NFT(G)$ graph from Fig. 7.11. In contrast to the procedure *NFT_REDUCTION*, the *NFT_VERIFY* has a timing complexity of $O(n^k)$, which increases exponentially with the node factor k.

The combination of the analytical and exhaustive search method is utilized to find a locally optimal G^*_{opt}. It is interesting that no edge among the 23 edges in G^* is further removable but still fulfill the requirement of $1 - NFT(G)$. Consequently, the $1 - NFT(G)$ graph in Fig. 7.11 indicates the locally optimal graph for task G.

7.2.4 Experiments for Node Fault Tolerance

To demonstrate the improvement of system-level reliability, the task with 9 nodes in Fig. 7.8 are mapped onto the network of 11 PEs. Tables 7.3 and 7.4 gives the results of the remapping scenarios under 1-NFT and 2-NFT respectively. It is found that under all cases of failure, at least one successful task remapping scenario is found.

Figure 7.12a and b present the task graph and corresponding 1-NFT PE network. Figure 7.12c and d visualize two scenarios of 1-NFT based task mapping. The mapping reflects the schemes is Table 7.3. The redundant edges are labelled as thin lines. No redundant edge exists for the scenario in Fig. 7.12d, which verifies that the network in Fig. 7.12b is *locally optimal 1-NFT*. This indicates that no further edge in Fig. 7.12b is removable.

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

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

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

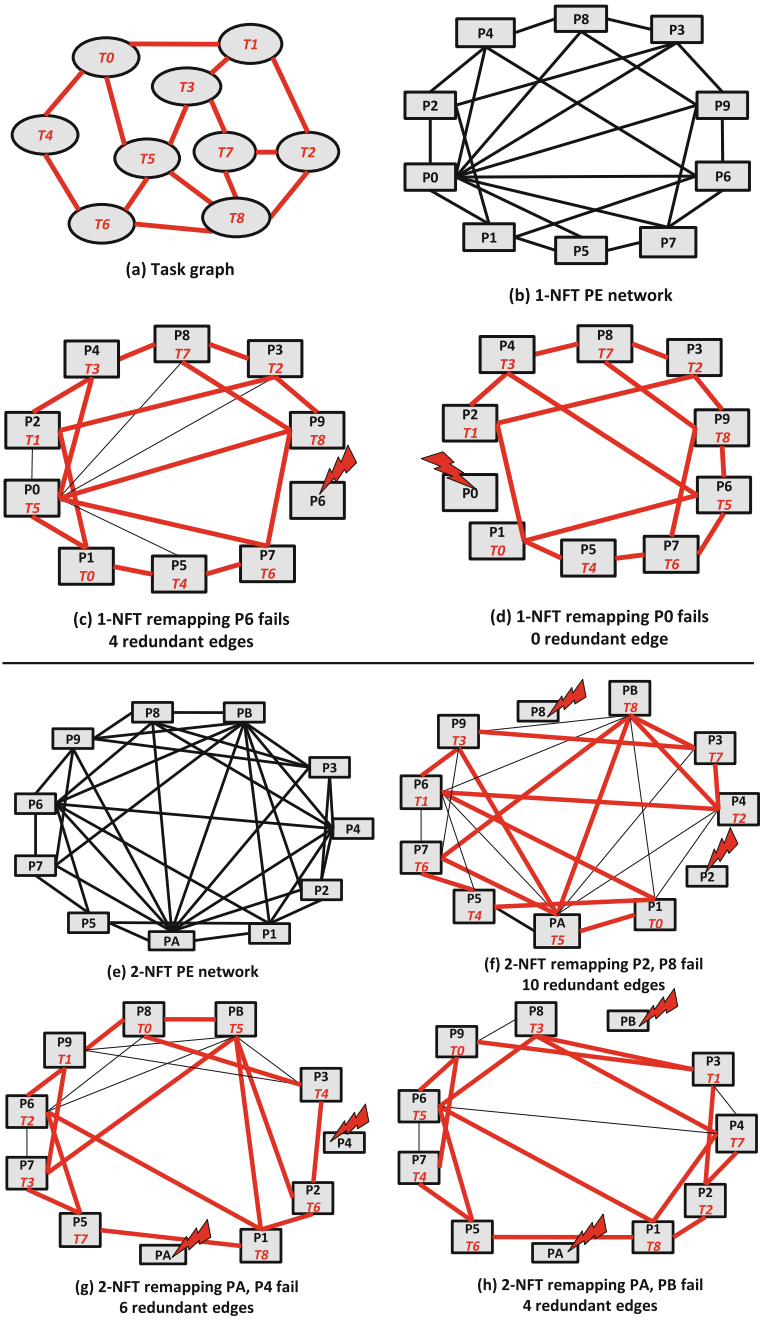


Fig. 7.12 NFT mapping schemes with one and two fail cores [204] Copyright ©2016 **IEEE**

Figure 7.12e presents the 2-NFT PE network from the task graph. Out of three failure scenarios shown in Fig. 7.12f–h, the minimal number of redundant edges is 4. This implies that the network in Fig. 7.12e is not locally optimal 2-NFT, which can be further optimized through the procedure of edge reduction in Algorithm 5.

Finally, tasks are assigned with real operators and execute them using the system-level reliability exploration framework in Sect. 7.1. Figure 7.13 presents a virtual prototype of several ARM processors as processing elements (PEs) and one task manager. Failure injection into the PEs is realized by occasionally falling the execution of PE into an infinite loop to mimic the state of unresponsiveness. The firmware on central controller detects such situation with the help of PE's watchdog timer and starts the task remapping procedure as described in Sect. 7.1.1.2.

Figure 7.14a shows the directed task graph which exactly match the graph pattern in Fig. 7.8. The corresponding operators assigned to tasks are indicated in Fig. 7.14b. The example uses simple operators to avoid the long simulation time. 100 data tokens are streamed as input to task 4 and the results are read out from task 1.

The task remapping is statically implemented by storing Tables 7.3 and 7.4 as Look-up-Table into the memory of managers so that different mapping patterns can be found in the small and similar amount of time. Figure 7.14c and d present 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 the injection of failures and produce correct data outputs, regardless of the locations of failure.

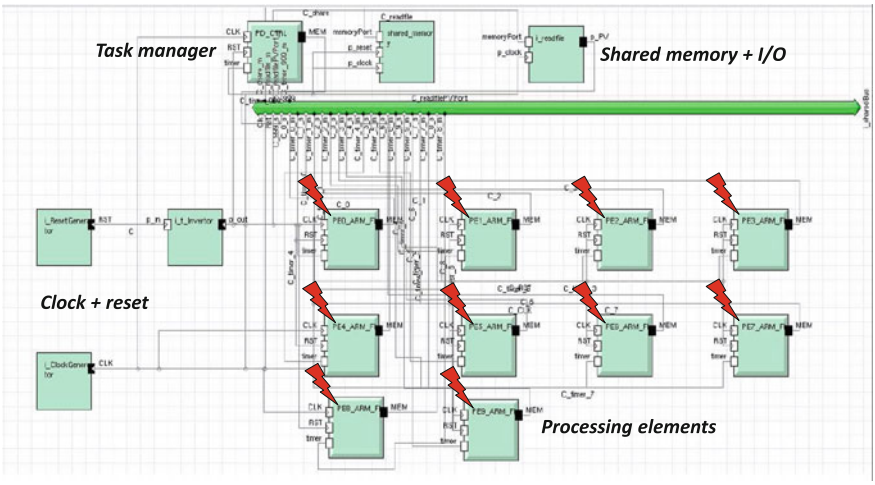
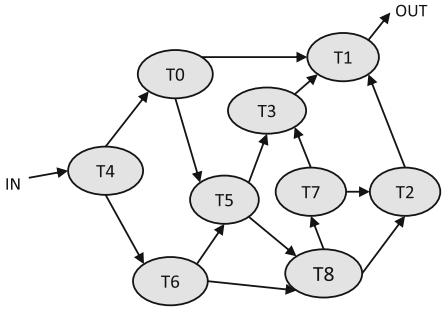


Fig. 7.13 Virtual prototype for NFT exploration [204] Copyright ©2016 **IEEE**



(a) Directed process network

Node	Operator
T0	in1 << 2
T1	OUT = in1 * in2 * in3
T2	in1 & in2
T3	in1 ^ in2
T4	IN >> 2
T5	in1 | in2
T6	in1 mod 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

Fig. 7.14 Task execution time under 1-NFT and 2-NFT [204] Copyright ©2016 **IEEE**

7.2.5 Summary

In this work, a divide-and-conquer methodology based on the theory of Node Fault Tolerance is proposed to construct the fault tolerant network topology for generic KPN style tasks with relatively small amount of redundant edges. Exhaustive search based graph verification and reduction algorithms are introduced to further reduce the number of redundant edges. Real KPN tasks on a multiprocessor virtual prototype are utilized to verify the correctness of proposed methodology.

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 book, 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 have 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. The fault injection is extended with dynamic timing analysis to evaluate the impacts of timing errors to the applications. A power/thermal/logic delay co-simulation framework is presented for integrating fault injection with the 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

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6_8

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 exist underutilized resources, which incurs a 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 the 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 cores 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 the construction of an ad-hoc fault tolerant network for arbitrary task graph with the optimal amount of connecting edges is presented and verified using exhaustive search based algorithm.

8.2 Outlook

The techniques proposed in this book 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 the 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 aging 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, the high-level synthesizer can select hardware modules with sufficient reliability level according to user's constraints. The 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. The 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 book 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 is 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 supercomputing. Instead of aggressive task migration in a complex network of processing elements which imposes large processing power, the ad-hoc NFT network with a relatively smaller amount of edges can guarantee the functional state facing predefined amount of failure cores. A partner idea focusing on Edge Fault Tolerance is also worth investigating for arbitrary process networks.

Curriculum Vitae

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

Since Jan. 2017	Assistant Professor at Shenzhen Institute of Advanced Integration Technology (SIAIT) Chinese Academy of Sciences (CAS) and the Chinese University of Hong Kong (CUHK), China
Oct. 2015 – Dec. 2016	Research fellow at School of Electrical and Electronic Engineering Nanyang Technological University, Singapore
Sept. 2010 – Sept. 2015	Research associate at Institute for Communication Technologies and Embedded Systems (ICE) RWTH-Aachen University, Germany PhD dissertation "High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip"
Oct. 2007 – Sept. 2009	Master student 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 student at Department of Physics Shanghai Jiao Tong University, China

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6

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

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6

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

Bibliography

1. Chattopadhyay A, Meyr H, Leupers R (2008) LISA: a uniform ADL for embedded processor modelling, implementation and software toolsuite generation, Chap. 5. Morgan Kaufmann, San Francisco, pp 95–130
2. Kahng A, Kang S, Kumar R, Sartori J (2010) Designing processors from the ground up to allow voltage/reliability tradeoffs. In: HPCA (2010)
3. Aguirre MA, Tombs JN, Baena V, Muñoz-Chavero F, Torralba A, Fernández-León A, Tortosa F (2005) Ft-unshades: A new system for seu injection, analysis and diagnostics over post synthesis netlist. In: Proceedings of the NASA military and aerospace programmable logic devices, MAPLD, Washington, DC
4. Akers SB, Krishnamurthy B (1987) On group graphs and their fault tolerance. IEEE Trans Comput 100(7):885–888
5. Alam MA, Mahapatra S (2005) A comprehensive model of PMOS NBTI degradation. Microelectron Reliab 45(1):71–81
6. Alderighi M, Casini F, D'Angelo S, Mancini M, Codinachs DM, Pastore S, Poivey C, Sechi GR, Sorrenti G, Weigand R (2010) Experimental validation of fault injection analyses by the flipper tool. IEEE Trans Nucl Sci 4(57):2129–2134
7. Alexandrescu D, Costenaro E, Nicolaidis M (2011) A practical approach to single event transients analysis for highly complex designs. IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT). IEEE, New York, pp 155–163
8. Arlat J, Crouzet Y, Laprie J-C (1989) Fault injection for dependability validation of fault-tolerant computing systems. In: FTCS. pp 348–355
9. http://www.arm.com/products/processors/classic/arm9/index.php, ARM
10. Association SI et al (2003) International technology roadmap for semiconductors (ITRS), 2003rd edn. Hsinchu, Taiwan
11. Austin T et al., (1999) DIVA: a reliable substrate for deep submicron microarchitecture design. In: International Symposium on Microarchitecture, pp 196–207
12. Avizienis A, Kelly JP (1984) Fault tolerance by design diversity: concepts and experiments. Computer 17(8):67–80
13. Avizienis A, Chen L (1977) On the implementation of n-version programming for software fault tolerance during execution. Proc IEEE COMPSAC 77:149–155
14. Baek W, Chilimbi TM (2010) Green: a framework for supporting energy-conscious programming using controlled approximation. ACM sigplan notices, vol 45, edn 6. ACM, New York, pp 198–209

© Springer Science+Business Media Singapore 2018

Z. Wang and A. Chattopadhyay, *High-level Estimation and Exploration of Reliability for Multi-Processor System-on-Chip*, Computer Architecture and Design Methodologies, DOI 10.1007/978-981-10-1073-6

15. Banerjee N, Karakonstantis G, Roy K (2007) Process variation tolerant low power DCT architecture. In: Proceedings of the conference on Design, automation and test in Europe. EDA Consortium, pp 630–635
16. Baraza JC, Gracia J, Gil D, Gil PJ (2002) A prototype of a VHDL-based fault injection tool: description and application. J Syst Architect 47(10):847–867
17. Beltrame G, Fossati L, Sciuto D (2009) Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration. IEEE Trans Comput Aided Des Integr Circuits Syst 28(12):1857–1869
18. Berrojo L, Corno F, Reorda MS, Squillero G, Entrena L, Lopez C (2002) New techniques for speeding-up fault-injection campaigns. Proceedings of the design, automation and test in Europe conference and exhibition. IEEE, New York, pp 847–852
19. Bhardwaj S, Wang W, Vattikonda R, Cao Y, Vrudhula S (2006) Predictive modeling of the NBTI effect for reliable design. Custom integrated circuits conference (2006) CICC'06. IEEE. IEEE, New York, pp 189–192
20. BIOS A (2007) Kernel developer's guide for amd npt family 0fh processors. http://support.amd.com/us/Processor_TechDocs/32559.pdf
21. Biswas A, Racunas P, Emer JS, Mukherjee SS (2007) Computing accurate AVFs using ACE analysis on performance models: a rebuttal. Comput Architect Lett 7(1):21–24
22. Blume H, Becker D, Botteck M, Brakensiek J, Noll T (2006) Hybrid functional and instruction level power modeling for embedded processors. In: Embedded computer systems: architectures, modeling, and simulation, pp 216–226
23. Borkar S (2010) The exascale challenge. In: Proceedings of the VLSI-DAT
24. Boué J, Pétillon P, Crouzet Y, Mefisto-l: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In: Twenty-eighth annual international symposium on fault-tolerant computing, digest of papers. IEEE, New York, pp 168–173
25. Bowman K et al., (2011) A 45 nm resilient microprocessor core for dynamic variation tolerance. In: IEEE JSCC, pp 194–208
26. AES implementation in C. http://gladman.plushost.co.uk/oldsite/, Brian Gladman
27. Shannon CE (1949) Communication in the presence of noise. Proc Inst Radio Eng 37:10–21
28. Castrillon J, Zhang D, Kempf T, Vanthournout B, Leupers R, Ascheid G (2009) Task management in MPSOCS: an ASIP approach, series, ICCAD'09
29. Chang K-J, Chen Y-Y, (2007) System-level fault injection in system design platform. In: ISIS 2007 Proceedings of the 8th symposium on advanced intelligent systems, pp 354–359
30. Cheng Y, Anguo M, Zhang M (2012) Accurate and simplified prediction of l2 cache vulnerability for cost-efficient soft error protection. IEICE Trans Inf Syst 95(1):56–66
31. Chippa V, Raghunathan A, Roy K, Chakradhar S (2011) Dynamic effort scaling: managing the quality-efficiency tradeoff. Proceedings of the 48th design automation conference. ACM, New York, pp 603–608
32. Chippa VK, Chakradhar ST, Roy K, Raghunathan A (2013) Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of the 50th annual design automation conference. ACM, New York, p 113
33. Chippa VK, Mohapatra D, Raghunathan A, Roy K, Chakradhar ST (2010) Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. Proceedings of the 47th design automation conference. ACM, New York, pp 555–560
34. Chippa VK, Venkataramani S, Chakradhar ST, Roy K, Raghunathan A (2013) Approximate computing: an integrated hardware approach. In: IEEE Asilomar, pp 111–117
35. Cho H, Mirkhani S, Cher C-Y, Abraham JA, Mitra S (2013) Quantitative evaluation of soft error injection techniques for robust system design. In: Design automation conference (DAC), pp 1–10
36. Constantin J et al., (2015) Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In: IEEE DATE, pp 381–386
37. Constantin J, Burg A, Wang Z, Chattopadhyay A, Karakonstantis G (2016) "Statistical fault injection for impact-evaluation of timing errors on application performance. In: Proceedings of the 53rd annual design automation conference. ACM, New York, p 13

38. Constantin J, Wang L, Karakonstantis G, Chattopadhyay A, Burg A (2015) Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In: Proceedings of the 2015 design, automation and test in Europe conference and exhibition. EDA Consortium, 2015, pp 381–386
39. Constantinescu C (2003) Trends and challenges in VLSI circuit reliability. IEEE Micro 23(4):14–19
40. Cormen TH (2009) Introduction to algorithms. MIT press, Cambridge 2009
41. Coskun AK, Rosing TS, Gross KC (2008) Temperature management in multiprocessor SOCS using online learning, series, DAC'08
42. Cox DR, Hinkley DV (1979) Theoretical statistics. CRC Press, Boca Raton
43. Brooks D, Tiwari V, Martonosi M (2000) Wattch: a framework for architectural-level power analysis and optimizations. Proceedings of the 27th annual international symposium on computer architecture, ISCA '00. ACM, New York, pp 83–94
44. Kammler D, Guan J, Ascheid G, Leupers R, Meyr H (2009) A fast and flexible platform for fault injection and evaluation in verilog-based simulations. In: Proceedings of the 3rd IEEE international conference on secure software integration and reliability improvement (SSIRI)
45. Daemen J, Rijmen V (2002) The design of rijndael: AES - the advanced encryption standard. Springer, Berlin
46. Das A, Kumar A, Veeravalli B (2013) Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In: Conference on design, automation and test in Europe, series, DATE'13
47. DeHon A, Quinn HM, Carter NP (2010) Vision for cross-layer optimization to address the dual challenges of energy and reliability. DATE. IEEE, New York, pp 1017–1022
48. Dell TJ (1997) A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectron Div, 1–23
49. Derin O, Kabakci D, Fiorin L (2011) Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In: Fifth ACM/IEEE international symposium on networks-on-chip, series, NOCS'11
50. Dirk L, Nelson ME, Ziegler JF, Thompson A, Zabel TH (2003) Terrestrial thermal neutrons. IEEE Trans Nucl Sci 50(6):2060–2064
51. Donald J, Martonosi M (2006) Techniques for multicore thermal management: Classification and new exploration. ACM SIGARCH Computer Architecture News, vol 34, edn 2. IEEE Computer Society, New York, pp 78–88
52. Dong X., Muralimanohar N, Jouppi N, Kaufmann R, Xie Y (2009) Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In: Proceedings of the conference on high performance computing networking, storage and analysis. ACM, New York, p 57
53. Dubrova E (2008) Fault tolerant design: an introduction. Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden
54. Berlekamp E, McEliece R, van Tilborg H (1978) On the Inherent intractability of certain coding problems. IEEE Trans Inf Theory 24(3):384–386
55. Normand E (1996) Single event upset at ground level. IEEE Trans Nucl Sci 43(6):2742–2750
56. Emer J, Ahuja P, Borch E, Klauser A, Luk C-K, Manne S, Mukherjee SS, Patil H, Wallace S, Binkert N et al (2002) ASIM: a performance model framework. Computer 35(2):68–76
57. Emre Y, Chakrabarti C (2013) Techniques for compensating memory errors in JPEG2000. IEEE Trans VLSI Syst, 21(1):159–163. http://dx.doi.org/10.1109/TVLSI.2011.2180407
58. Entrena L, López C, Olías E (2001) Automatic generation of fault tolerant VHDL designs in RTL. In: FDL (Forum on design languages). Citeseer
59. Ernst D, Kim NS, Das S, Pant S, Rao R, Pham T, Ziesler C, Blaauw D, Austin T, Flautner K (2003) Razor: a low-power pipeline based on circuit-level timing speculation. Proceedings of 36th annual IEEE/ACM international symposium on microarchitecture, MICRO-36. IEEE, New York, pp 7–18
60. Esmaeilzadeh H, Sampson A, Ceze L, Burger D (2012) Architecture support for disciplined approximate programming. ACM SIGPLAN notices, vol 47, edn 4. ACM, New York, pp 301–312

61. UMC free library 90 nm process. http://freelibrary.faraday-tech.com/, Faraday
62. Faruque M, Dinavahi V, Steurer M, Monti A, Strunz K, Martinez J, Chang G, Jatskevich J, Iravani R, Davoudi A (2012) Interfacing issues in multi-domain simulation tools. IEEE Trans Power Del 27(1):439–448
63. Fernandez-Alonso E, Castells-Rufas D, Joven J, Carrabina J (2012) Survey of NOC and programming models proposals for MPSOC. Int J Comput Sci Iss 9(2):22–32
64. Folkesson P, Svensson S, Karlsson J (1998) A comparison of simulation based and scan chain implemented fault injection. Twenty-eighth annual international symposium on fault-tolerant computing. Digest of papers. IEEE, New York, pp 284–293
65. Fuchs E (1996) An evaluation of the error detection mechanisms in mars using software-implemented fault injection. Dependable computing-EDCC-2. Springer, Berlin, pp 73–90
66. Georgakos G, Huber P, Ostermayr M, Amirante E, Ruckerbauer F (2007) Investigation of increased multi-bit failure rate due to neutron induced SEU in advanced embedded SRAMS. In: 2007 IEEE symposium on VLSI circuits
67. George NJ, Elks CR, Johnson BW, Lach J (2010) Transient fault models and avf estimation revisited. 2010 IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, New York, pp 477–486
68. Gill B, Seifert N, Zia V (2009) Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node. IEEE international reliability physics symposium. IEEE, New York, pp 199–205
69. Gilles K (1974) The semantics of a simple language for parallel programming. In: Information processing'74: Proceedings of the IFIP congress, vol 74, pp 471–475
70. Gonzalez RC (2009) Digital image processing. Pearson Education, India
71. Gordon MS, Goldhagen P, Rodbell KP, Zabel T, Tang H, Clem J, Bailey P (2004) Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground. IEEE Trans Nucl Sci 51(6):3427–3434
72. Gupta V, Mohapatra D, Park SP, Raghunathan A, Roy K (2011) Impact: imprecise adders for low-power approximate computing. Proceedings of the 17th IEEE/ACM international symposium on low-power electronics and design. IEEE Press, New York, pp 409–414
73. Guthaus M , Ringenberg J, Ernst D, Austin T, Mudge T, Brown R, 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, New York, pp 3–14. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=990739
74. Cho H, Leem L, Mitra S (2012) ERSA: error resilient system architecture for probabilistic applications. IEEE Trans CAD Integr Circuits Syst 31(4):546–558
75. Keding MWH, Hürtgen F, Coors M (1998) Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach. In: Proceedings of the international conference on signal processing application and technology (ICSPAT), Toronto
76. Haghdoost A, Asadi H, Baniasadi A (2010) System-level vulnerability estimation for data caches. IEEE 16th Pacific Rim international symposium on dependable computing (PRDC. IEEE, New York, pp 157–164
77. Handke D (1999) Graphs with distance guarantees, PhD dissertation
78. Harary F, Hayes JP (1993) Edge fault tolerance in graphs. Networks 23(2):135–142
79. Harary F, Hayes JP (1996) Node fault tolerance in graphs. Networks 27(1):19–23
80. Hartman AS, Thomas DE (2012) Lifetime improvement through runtime wear-based task mapping, series, CODES+ISSS'12
81. Hitchcock RB Sr (1982) Timing verification and the timing analysis program. Proceedings of the 19th design automation conference. IEEE Press, New York, pp 594–604
82. Hosseinabady M, Kakoee MR, Mathew J, Pradhan DK (2007) Reliable network-on-chip based on generalized de bruijn graph. IEEE international high level design validation and test workshop, HLVDT 2007. IEEE, New York, pp 3–10
83. Documentation. http://lava.cs.virginia.edu/HotSpot/documentation.htm, HotSpot 6.0
84. Hsieh M-Y, Rodrigues A, Riesen R, Thompson K, Song W (2011) A framework for architecture-level power, area, and thermal simulation and its application to network-on-chip design exploration. ACM SIGMETRICS Perform Eval Rev 38(4):63–68

85. Huang T, Yang G, Tang G (1979) A fast two-dimensional median filtering algorithm. IEEE Trans Acoust Speech Signal Process 27(1):13–18
86. Huang W, Ghosh S, Sankaranarayanan K, Skadron K, Stan MR (2005) Hotspot: thermal modeling for CMOS VLSI systems. In: IEEE Transactions on component packaging and manufacturing technology, pp 200–205
87. Huynh-Thu Q, Ghanbari M (2008) Scope of validity of psnr in image/video quality assessment. Electron Lett 44(13):800–801
88. Ibe E, Taniguchi H, Yahagi Y, Shimbo K, Toba T (2009) Scaling effects on neutron-induced soft error in SRAMS down to 22 nm process. In: Third workshop on dependable and secure nanocomputing, 2009
89. Ibe E, Taniguchi H, Yahagi Y, Shimbo K-I, Toba T (2010) Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. IEEE Trans Electron Dev 57(7):1527–1538
90. Clark JA, Pradhan DK (1995) Fault injection: a method for validating computer-system dependability. IEEE Comput 28(6):47–56
91. Srinivasan J, Adve SV, Bose P, Rivers JA (2004) The case for lifetime reliability-aware microprocessors. SIGARCH Comput Archit News 32(2):276
92. Jeitler M, Delvai M, Reichör S (2009) Fuse-a hardware accelerated HDL fault injection tool. 5th southern conference on programmable logic, 2009. SPL. IEEE, New York, pp 89–94
93. Jenn E, Arlat J, Rimén M, Ohlsson J, Karlsson J (1994) Fault injection into VHDL models: the mefisto tool. FTCS 1994:66–75
94. JESD89A JS (2006) Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices. JEDEC Solid State Technol Assoc
95. Amir K, Eric B (2001) 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
96. Kahn G (1974) The semantics of simple language for parallel programming. In: IFIP Congress'74
97. Kahng AB, Kang S (2012) Accuracy-configurable adder for approximate arithmetic designs. Proceedings of the 49th annual design automation conference. ACM, New York, pp 820–825
98. Kanda K, Nose K, Kawaguchi H, Sakurai T (2001) Design impact of positive temperature dependence on drain current in sub-1-v CMOS VLSIs. IEEE J Solid State Circuits 36(10):1559–1564
99. Karakonstantis G, Mohapatra D, Roy K (2012) Logic and memory design based on unequal error protection for voltage-scalable, robust and adaptive DSP systems. Signal Process Syst 68(3):415–431
100. Karlsson J, Liden P, Dahlgren P, Johansson R, Gunneflo U (1994) Using heavy-ion radiation to validate fault-handling mechanisms. IEEE Micro 1:8–11
101. Kiusalaas J (2010) Numerical methods in engineering with MATLAB®. Cambridge University Press, Cambridge
102. Koren I, Krishna CM (2010) Fault-tolerant systems. Morgan Kaufmann, San Francisco
103. Krewell K (2001) Intel's mckinley comes into view. Microprocess Rep 15(10):1
104. Kulkarni P, Gupta P, Ercegovac M (2011) Trading accuracy for power with an underdesigned multiplier architecture. 2011 24th international conference on VLSI design (VLSI design). IEEE, New York, pp 346–351
105. Myint L, Supnithi P (2012) Unequal error correction strategy for magnetic recording systems with multi-track processing. J Appl Phys 111
106. Lala JH, Harper RE (1994) Architectural principles for safety-critical real-time applications. Proc IEEE 82(1):25–40
107. Lampret D, Chen C-M, Mlinar M, Rydberg J, Ziv-Av M, Ziomkowski C, McGary G, Gardner B, Mathur R, Bolado M (2003) Openrisc 1000 architecture manual. Descr Assembl Mnemon OR1200

108. Lazowska ED, Zahorjan J, Graham GS, Sevcik KC (1984) Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall Inc, Englewood Cliffs
109. Lee C, Kim H, Park H, Kim S, Oh H, Ha S (2010) A task remapping technique for reliable multi-core embedded systems, series, CODES/ISSS'10
110. Lee I, Kwon J, Park J, Park J (2013) Priority based error correction code (ECC) for the embedded SRAM memories in H.264 system. In: Signal processing systems, vol 73, edn 2, pp 123–136. http://dx.doi.org/10.1007/s11265-013-0736-4
111. Lee J, Shrivastava A (2010) A compiler-microarchitecture hybrid approach to soft error reduction for register files. IEEE Trans CAD of Integr Circuits Syst 29(7):1018–1027
112. Leupers R, Temam O (2010) Processor and system-on-chip simulation. Springer, Berlin
113. Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2009) MCPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. 42nd annual IEEE/ACM international symposium on microarchitecture, (2009) MICRO-42. IEEE, New York, pp 469–480
114. Li S, Chen K, Hsieh M-Y, Muralimanohar N, Kersey CD, Brockman JB, Rodrigues AF, Jouppi NP, (2011) System implications of memory reliability in exascale computing. In: Proceedings of the international conference for high performance computing, Networking, storage and analysis. ACM, New York, p 46
115. Lin S, Costello D Jr (1983) Error control coding: fundamentals and applications. Prentice Hall, Englewood Cliffs
116. Lingamneni A, Enz C, Nagel J-L, Palem K, Piguet C (2011) Energy parsimonious circuit design through probabilistic pruning. Design, automation and test in europe conference and exhibition (DATE). IEEE, New York, pp 1–6
117. Lingamneni A, Enz C, Palem K, Piguet C (2011) Parsimonious circuits for error-tolerant applications through probabilistic logic minimization. In: Integrated circuit and system design. power and timing modeling, optimization, and simulation. Springer, Berlin, pp 204–213
118. Littlewood B (1996) The impact of diversity upon common mode failures. Reliability engineering and system safety 51(1):101–113
119. Lu W, Radetzki M (2011) Efficient fault simulation of systemc designs. 2011 14th euromicro conference on digital system design (DSD). IEEE, New York, pp 487–494
120. Lyons RE, Vanderkulk W (1962) The use of triple-modular redundancy to improve computer reliability. IBM J Res Develop 6(2):200–209
121. Li M, Ramachandran P, Sahoo SK, Adve SV, Adve VS, Zhou Y (2008) Understanding the propagation of hard errors to software and implications for resilient system design. In: Proceedings of the international conference on architectural support for programming languages and operating systems (ASPLOS)
122. Michael M, Große D, Drechsler R (2011) Analyzing dependability measures at the electronic system level. In: Forum on specification and design languages, series, FDL'11
123. Sugihara M, Ishihara T, Hashimoto K, Muroyama M (March 2006) A simulation-based soft error estimation methodology for computer systems. In: 7th international symposium on quality. Electronic design 2006
124. Madeira H, Rela M, Moreira F, Silva JG (1994) Rifle: a general purpose pin-level fault injector. Dependable computing–EDCC-1. Springer, Berlin, pp 197–216
125. Mansour W, Velazco R (2013) An automated SEU fault-injection method and tool for HDL-based designs. IEEE Trans Nucl Sci 60(4):2728–2733
126. Mansour W, Velazco R (2013) Seu fault-injection in VHDL-based processors: a case study. J Electron Test 29(1):87–94
127. Meixner A, Bauer ME, Sorin MD (2007) Argus: Low-cost, comprehensive error detection in simple cores, series, MICRO 40
128. Miao J, Gerstlauer A, Orshansky M (2013) Approximate logic synthesis under general error magnitude and frequency constraints. 2013 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, New York, pp 779–786

129. Miao J, He K, Gerstlauer A, Orshansky M (2012) Modeling and synthesis of quality-energy optimal approximate adders. Proceedings of the international conference on computer-aided design. ACM, New York, pp 728–735
130. Mirkhani S, Cho H, Mitra S, Abraham JA (2014) Rethinking error injection for effective resilience. 19th Asia and South Pacific design automation conference (ASP-DAC). IEEE, New York, pp 390–393
131. Mirkhani S, Mitra S, Cher C-Y, Abraham J (2015) Efficient soft error vulnerability estimation of complex designs. In: Proceedings of the 2015 design, automation and test in Europe conference and exhibition. EDA consortium, pp 103–108
132. Misera S, Vierhaus HT, Sieber A (2007) Fault injection techniques and their accelerated simulation in systemc. 10th Euromicro conference on digital system design architectures, methods and tools, DSD 2007. IEEE, New York, pp 587–595
133. Mitra S, Saxena NR, McCluskey EJ (2002) A design diversity metric and analysis of redundant systems. IEEE Trans Comput 51(5):498–510
134. Mitra S, Saxena NR, McCluskey EJ (2004) Efficient design diversity estimation for combinational circuits. IEEE Trans Comput 53(11):1483–1492
135. Montesinos P, Liu W, Torrellas J (2007) Using register lifetime predictions to protect register files against soft errors. 37th annual IEEE/IFIP international conference on dependable systems and networks, DSN'07. IEEE, New York, pp 286–296
136. Moore GE (2006) Cramming more components onto integrated circuits, vol 38, edn 8 1965, pp 114 ff. (IEEE Solid-State Circuits Newsl, 3(20):33–35, 2006)
137. Mukherjee SS, Kontz M, Reinhardt SK (2002) Detailed design and evaluation of redundant multi-threading alternatives. In: Proceedings of 29th annual international symposium on computer architecture, pp 99–110
138. Mukherjee SS, Weaver CT, Emer JS, Reinhardt SK, Austin TM (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: MICRO, pp 29–42
139. Muralimanohar N, Balasubramonian R, Jouppi NP (2009) Cacti 6.0: A tool to model large caches. In: HP laboratories, pp 22–31
140. Omana M, Papasso G, Rossi D, Metra C, A model for transient fault propagation in combinatorial logic. In: 9th IEEE on-line testing symposium, IOLTS 2003. IEEE, New York, pp 111–115
141. Palem K Lingamneni A (2013) Ten years of building broken chips: the physics and engineering of inexact computing. ACM Trans Embed Comput Syst 12(2s):87:1–87:23. http://doi.acm.org/10.1145/2465787.2465789
142. Park Y, Pasricha S, Kurdahi F, Dutt N (2011) A multi-granularity power modeling methodology for embedded processors. IEEE Trans Very Large Scale Integr (VLSI) Syst 19(4):668–681
143. Paul S, Cai F, Zhang X, Bhunia S (2011) Reliability-driven ECC allocation for multiple bit error resilience in processor cache. IEEE Trans Comput 60(1):20–34
144. Peterson WW, Weldon EJ (1972) Error-correcting codes. MIT press, Cambridge
145. Processor designer: C compiler reference manual 2013.06
146. Hegde R, Shanbhag NR, Energy-efficient signal processing via algorithmic noise-tolerance, series, ISLPED'99
147. Naseer R (2008) parallel double error correcting code design to mitigate multi-bit upsets in SRAMs, In: ESSCIRC, pp 222–225
148. Hamming RW (1950) Error detecting and error correcting codes. Bell Syst Tech J XXVI 2:147–160
149. Rahbaran B, Steininger A, Handl T (2004) Built-in fault injection in hardware-the fidyco example. Proceedings of 2004 IEEE international conference on field-programmable technology, 2004. IEEE, New York, pp 327–332
150. Rajendiran A, Ananthanarayanan S, Patel HD, Tripunitara MV, Garg S (2012) Reliable computing with ultra-reduced instruction set co-processors. In: Proceedings of the 49th design automation conference, pp 697–702

151. Rákossy ZE, Merchant F, Acosta-Aponte A, Nandy S, Chattopadhyay A (2014) Efficient and scalable CGRA-based implementation of column-wise givens rotation. In: 25th international conference on application-specific systems, architectures and processors (ASAP), pp 188–189
152. Ranjan A, Raha A, Venkataramani S, Roy K, Raghunathan A (2014) Aslan: synthesis of approximate sequential circuits. In: Proceedings of the conference on design, automation and test in Europe. European Design and Automation Association, p 364
153. Rehman S, Shafique M, Kriebel F, Henkel J (2011) Reliable software for unreliable hardware: embedded code generation aiming at reliability. In: CODES+ISSS, pp 237–246
154. Rehman S, Toma A, Kriebel F, Shafique M, Chen J-J, Henkel J (2013) Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. IEEE 19th real-time and embedded technology and applications symposium (RTAS). IEEE, New York, pp 273–282
155. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) Swift: Software implemented fault tolerance. In: Proceedings of the international symposium on code generation and optimization, pp 243–254
156. Riter R (1995) Modeling and testing a critical fault-tolerant multi-process system. In: Fault-tolerant computing, FTCS-25, pp 516–521
157. E. Rotenberg, AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In: FTCS. IEEE Computer Society Press, New York, pp 84–93
158. Ahmed S (2011) Unequal error protection using fountain codes with applications to video communication. IEEE Trans Multimed 13(1):92–101
159. Baloch S, Arslan T, Stoica A (2005) Efficient error correcting codes for on-chip DRAM applications for space missions. In: IEEE aerospace, pp 1–9
160. Reinhardt SK, Mukherjee SS (2000) Transient fault detection via simultaneous multithreading, series, ISCA'00, pp 25–36
161. Krishnaswamy S, Viamontes GF, Markov IL, Hayes JP (2005) 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, New York, pp 282–287
162. Krishnaswamy S, Viamontes GF, Markov IL, Hayes JP (2008) Probabilistic transfer matrices in symbolic reliability analysis of logic circuits. ACM Trans Des Autom Electr Syst 13(1):
163. Rehman S, Shafique M, Henkel J (2012) Instruction scheduling for reliability-aware compilation, series, DAC'12. ACM, New York, pp 1292–1300. http://doi.acm.org/10.1145/2228360.2228601
164. Rehman S, Shafique M, Kriebel F, Henkel J (2012) RAISE: Reliability-Aware Instruction SchEduling for unreliable hardware. In: ASP-DAC'12, 30 2012-Feb. 2 2012, pp 671 –676
165. Sakurai T et al (1990) Alpha-power law mosfet model and its applications to CMOS inverter delay and other formulas. IEEE J Solid State Circuits 25(2):584–594
166. Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L, Grossman D (2011) Enerj: Approximate data types for safe and general low-power computation. ACM SIGPLAN notices, vol 46, edn 6. ACM, New York, pp 164–174
167. Sassone A, Calimera A, Macii A, Macii E, Poncino M, Goldman R, Melikyan V, Babayan E, Rinaudo S (2012) Investigating the effects of inverted temperature dependence (ITD) on clock distribution networks. In: 2012 design. automation and test in Europe conference and exhibition, DATE 2012, pp 165–166
168. Schor L, Bacivarov I, Rai D, Yang H, Kang S, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems, series, CASES'12
169. Semiconductor T (2004) Soft errors in electronic memory-a white paper
170. Shafik RA, Rosinger P, Al-Hashimi BM (2008) Systemc-based minimum intrusive fault injection technique with improved fault representation. 14th IEEE international on-line testing symposium, IOLTS'08. IEEE, New York, pp 99–104
171. Shafique M, Rehman S, Aceituno PV, Henkel J (2013) Exploiting program-level masking and error propagation for constrained reliability optimization. In: Proceedings of the 50th annual design automation conference. ACM, New York, p 17

172. Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L (2002) Modeling the effect of technology trends on the soft error rate of combinational logic. Proceedings of the international conference on dependable systems and networks, (2002) DSN 2002. IEEE, New York, pp 389–398
173. Sieh V, Tschache O, Balbach F (1997) Verify: Evaluation of reliability using VHDL-models with embedded fault descriptions. Twenty-seventh annual international symposium on fault-tolerant computing, FTCS-27. Digest of papers. IEEE, New York, pp 32–36
174. Singh A, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends, series, DAC'13
175. Skadron K, Stan MR, Huang W, Velusamy S, Sankaranarayanan K, Tarjan D (2003) Temperature-aware microarchitecture. ACM SIGARCH Comput Architect News 31(2):2–13
176. Skorobogatov SP, Anderson RJ (2002) Optical fault induction attacks. In: Çetin Kaya Koç BSK Jr, Paar C (eds) CHES, series, Lecture Notes in Computer Science, vol 2523. Springer, Berlin, pp 2–12
177. Slayman C (2010) Soft errors-past history and recent discoveries. IEEE international integrated reliability workshop final report (IRW). IEEE, New York, pp 25–30
178. Somers J, Director F, Graham S (2002) Stratus FTserver–intel fault tolerant platform. In: Rap tech Intel Developer Forum, Fall
179. Soteriou V, Eisley N, Wang H, Li B, Peh L-S (2007) Polaris: a system-level roadmapping toolchain for on-chip interconnection networks. IEEE Trans Very Large Scale Integr (VLSI) Syst 15(8):855–868
180. Spainhower L, Gregg TA (1999) IBM s/390 parallel enterprise server G5 fault tolerance: a historical perspective. IBM J Res Develop 43(5.6): 863–873
181. Spielman DA (1996) Linear-time encodable and decodable error-correcting codes. IEEE Trans Inf Theory 42:388–397
182. Design compiler. http://www.synopsys.com/products/logic/design_compiler.html, Synopsys
183. Platform architect. http://www.synopsys.com/Prototyping/ArchitectureDesign, Synopsys
184. Processor designer. http://www.synopsys.com/Systems/BlockDesign/processorDev, Synopsys
185. Tang X, De VK, Meindl JD (1997) Intrinsic mosfet parameter fluctuations due to random dopant placement. IEEE Trans Very Large Scale Integr (VLSI) Syst 5(4):369–376
186. http://www.target.com, Target Compiler Technologies
187. Terraneo F, Zoni D, Fornaciari W (2015) 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, New York, p 5
188. CoSy compiler development system. www.ace.nl/compiler/cosy.html, The ACE Companies
189. Tiwari V, Malik S, Wolfe A (1994) Power analysis of embedded software: a first step towards software power minimization. IEEE Trans Very Large Scale Integr (VLSI) Syst 2(4):437–445
190. Tiwari V, Malik S, Wolfe A, Tien-Chien Lee M (1996) Instruction level power analysis and optimization of software. J VLSI Signal Process 13(2):223–238
191. Udipi AN, Muralimanohar N, Chatterjee N, Balasubramonian R, Davis A, Jouppi NP (2010) Rethinking dram design and organization for energy-constrained multi-cores. ACM SIGARCH Comput Architect News 38(3):175–186
192. Van Woudenberg JG, Witteman MF, Menarini F (2011) Practical optical fault injection on secure microcontrollers. In: Workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, New York 2011:91–99
193. Vangal SR, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, Finan D, Singh A, Jacob T, Jain S et al (2008) An 80-tile sub-100-w teraflops processor in 65-nm CMOS. IEEE J Solid State Circuits 43(1):29–41
194. Venkataramani S, Chippa VK, Chakradhar ST, Roy K, Raghunathan A (2013) Quality programmable vector processors for approximate computing. Proceedings of the 46th annual IEEE/ACM international symposium on microarchitecture. ACM, New York, pp 1–12

195. Venkataramani S, Roy K, Raghunathan A (2013) 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
196. Venkataramani S, Sabne A, Kozhikkottu V, Roy K, Raghunathan A (2012) Salsa: systematic logic synthesis of approximate circuits. Proceedings of the 49th annual design automation conference. ACM, New York, pp 796–801
197. Venkatesan R, Agarwal A, Roy K, Raghunathan A (2011) Macaco: modeling and analysis of circuits for approximate computing. Proceedings of the international conference on computer-aided design. IEEE Press, New York, pp 667–673
198. von Neumann J (1956) Probabilistic logics and synthesis of reliable organisms from unreliable components. In: McCarthy J (ed) Shannon C. Princeton University Press, Automata Studies, pp 43–98
199. Wang S (2011) Characterizing system-level vulnerability for instruction caches against soft errors. IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT). IEEE, New York, pp 356–363
200. Wang Z, Wang X, Chattopadhyay A, Rakosi ZE (2012) Asic synthesis using architecture description language. In: International symposium on VLSI design, automation, and test (VLSI-DAT), pp 1–4
201. Wang Z, Chen C, Sharma P, Chattopadhyay A (2014) System-level reliability exploration framework for heterogeneous MPSOC. Proceedings of the 24th edition of the great lakes symposium on VLSI. ACM, New York, pp 9–14
202. Wang Z, Karakonstantis G, Chattopadhyay A (2016) A low overhead error confinement method based on application statistical characteristics. Design, automation and test in Europe conference and exhibition (DATE). IEEE, New York, pp 1168–1171
203. Wang Z, Li R, Chattopadhyay A (2013) Opportunistic redundancy for improving reliability of embedded processors. In: 8th international design and test symposium, IDT (2013) Marrakesh. Morocco, pp 1–6
204. Wang Z, Littarru A, Ugwu E, Kanwal S, Chattopadhyay A (2016) Reliable many-core system-on-chip design using k-node fault tolerant graphs. In: IEEE computer society annual symposium on VLSI. IEEE, New York
205. Wang Z, Paul G, Chattopadhyay A (2014) Processor design with asymmetric reliability. IEEE computer society annual symposium on VLSI (ISVLSI). IEEE, New York, pp 565–570
206. Wang Z, Wang L, Xie H, Chattopadhyay A (2013) Power modeling and estimation during adl-driven embedded processor design. 2013 4th annual international conference on energy aware computing systems and applications (ICEAC). IEEE, New York, pp 97–102
207. Wang Z, Xie H, Chafekar S, Sai RUA, Chattopadhyay A (2015) Architectural error prediction using probabilistic error masking matrices. In: Asia symposium on quality electronic design (ASQED). IEEE, New York
208. Wang Z, Yang L, Chattopadhyay A (2015) Architectural reliability estimation using design diversity. 16th international symposium on quality electronic design (ISQED). IEEE, New York, pp 112–117
209. Weaver C, Austin T (2001) A fault tolerant approach to microprocessor design. International conference on dependable systems and networks, DSN 2001. IEEE, New York, pp 411–420
210. Weaver C, Emer J, Mukherjee SS, Reinhardt SK (2004) Techniques to reduce the soft error rate of a high-performance microprocessor. In: ACM SIGARCH computer architecture news, vol 32, edn 2. IEEE Computer Society, New York, p 264
211. Wilkerson C, Alameldeen AR, Chishti Z, Wu W, Somasekhar D, Lu S-L (2010) Reducing cache power with low-cost, multi-bit error-correcting codes. ACM SIGARCH Comput Architect News 38(3):83–93
212. Witte EM, Chattopadhyay A, Schliebusch O, Kammler D, Leupers R, Ascheid G, Meyr H (2005) Applying resource sharing algorithms to ADL-driven automatic asip implementation. ICCD 2005:193–199
213. Wood A, Jardine R, Bartlett W (2006) Data integrity in hp nonstop servers. In: Workshop on SELSE

214. Yueh W, Cho M, Mukhopadhyay S (2013) Perceptual quality preserving SRAM architecture for color motion pictures. In: Design, automation and test in Europe, DATE 13, Grenoble, France, March 18–22, 2013, pp 103–108. http://dl.acm.org/citation.cfm?id=2485315
215. Wang Z, Chen C, Chattopadhyay A (2013) Fast reliability exploration for embedded processors via high-level fault injection. In: ISQED, pp 265–272
216. Wang Z, Singh K, Chen C, Chattopadhyay A (2013) Accurate and efficient reliability estimation techniques during ADL-driven embedded processor design. DATE 2013:547–552
217. Wang Z, Li R, Chattopadhyay A (2013) Opportunistic redundancy for improving reliability of embedded processors. In: 8th IEEE international design and test symposium (IDT), 2013
218. Zarandi HR, Miremadi SG, Ejlali A (2003) Fault injection into verilog models for dependability evaluation of digital systems. In: null. IEEE, New York, p 281
219. Zhao W, Cao Y (2006) New generation of predictive technology model for sub-45 nm early design exploration. IEEE Trans Electron Dev 53(11):2816–2823
220. Zheng H, Fan L, Yue S (2008) Fitvs: A fpga-based emulation tool for high-efficiency hardness evaluation. International symposium on parallel and distributed processing with applications, ISPA'08. IEEE, New York, pp 525–531