Architectural framework for dynamically adaptable multiprocessors regarding aging, fault tolerance, performance and power consumption

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus-Senftenberg

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr. -Ing.)

genehmigte Dissertation

vorgelegt von

Mag. Sc.
Aleksandar Simevski
geboren am 26. Februar 1984 in Shtip

Gutachter: Prof. Dr.-Ing. Rolf Kraemer
Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus
Gutachter: Prof. Dr. Aristotel Tentov

Tag der mündlichen Prüfung: 27. November 2014

"I understand the world as a field for cultural contest among the nations."

"Ich verstehe die Welt als Feld für einen Wettkampf der Kulturen."

— Goce Delchev

Abstract

Despite the numerous benefits that Integrated Circuit (IC) technology downscaling brings, it also introduces many challenges. First of all, IC dependability is lowering: both lifetime reliability and resilience to single event effects is decreasing. Another major problem is the increased power consumption. On the other hand, the vast available space enables integrating hundreds of processor cores in a single chip! Multiprocessing is for over a decade the main architectural trend because of two reasons. Firstly, the performance of single processors gained by architectural innovations reached the upper limit i.e., the point of diminishing returns. Secondly, the operating frequency could not be increased due to the excessive power consumption, as pointed out. This work proposes a multiprocessor architectural framework that addresses many challenges related to dependability, power consumption and performance. The key idea is dynamical adaptation to the application requirements of fault tolerance and performance, which is possibly done at the lowest rates of aging and power dissipation. The application may select one of the three basic operating modes: de-stress, fault-tolerant and high-performance. De-stress mode prolongs multiprocessor lifetime and reduces power consumption by using core gating patterns that systematically power- or clock-off entire cores in the multiprocessor. These patterns use the information supplied by novel IC aging monitors. Fault-tolerant mode, on the other hand, increases error resilience by forming core-level NMR (N-modular redundant) systems using the multiprocessor cores. That is, entire cores are tightly synchronized to execute the same task simultaneously. Voting is done on each clock cycle using special, programmable NMR voters. Core-level NMR enables masking faults without invoking recovery procedures which is appreciated by timing-critical, or, real-time applications. Finally, high-performance mode is used for boosting multiprocessor performance. The framework is evaluated using a novel environment for automated fault injection, as well as a novel multiprocessor verification platform. A vast number of experiments were made which led to closed-form expressions that determine the number of cores N required to survive the projected mission time, given the fault rate. Moreover, a newly-developed method for lifetime evaluation based on the Weibul distribution shows the benefits of using core gating patterns. E.g., the new Youngest-First Round-Robin (YFRR) pattern enables up to 31% increase in system's lifetime compared to a simple Round-Robin.

Kurzfassung

Neben den vielzähligen Vorteilen, die die anhaltende Skalierung der Halbleitertechnologien mit sich bringt, gibt es auch eine Reihe von Herausforderungen. Insbesondere verringert sich die Zuverlässigkeit integrierter Schaltkreise (IC) durch die verstärkte Alterungseffekte aber auch durch erhöhte Anfälligkeit hochintegrierter Schaltungen auf Single-Event-Effekten. Ein weiteres Problem ist die steigende Leistungsaufnahme komplexer Schaltungen. Auf der anderen Seite erlaubt die Skalierung mittlerweile die Integration von Hunderten von Processorkernen in einem einzigen Chip. Seit mehr als einem Jahrzehnt ist dieses Prinzip des Multiprocessing aus zwei Gründen Trend bei der Architektur komplexer Prozessoren: Zum einen gibt es kaum noch Neuerungen in der Architektur von Single-Core-Prozessoren. Zum anderen kann die Taktfrequenz aufgrund der dadurch steigenden Leistungsaufnahme nicht weiter erhöht werden. Diese Arbeit stellt ein Framework zur Architektur von Multiprozessoren vor, dass sich der Herausforderungen bezüglich der Zuverlässigkeit, der Leistungsaufnahme und auch der Performance annimmt. Dabei ist die Kernidee die dynamische Anpassung an die Anforderungen der Anwendung bezüglich der Fehlertoleranz und der Leistungsfähigkeit. Dazu wählt die Anwendung einen der drei Hauptbetriebsmodi aus: *De-stress, Fault-Tolerance und High-Performance*. Der *De-stress-Modus* verlängert die Lebensdauer des Multiprozessors und reduziert die Leistungsaufnahme. Dazu werden mittels Clock-Gating oder durch Abschalten der Spannungsversorgung ganze Prozessorkerne abgeschaltet. Neu entwickelte Monitorschaltungen zur überwachung des Alterungsprozesses liefern dabei die Informationen, welcher Kern abgeschaltet werden sollte. Der *Fault-Tolerance-Modus* verringert die Fehleranfälligkeit durch Bildung einer N-modularen Redundanz auf Prozessorkernebene. Dazu werden mehrere Kerne so mit einander synchronisiert, dass sie exakt die selben Instruktionen durchführen. Ein programmierbarer NMR-Voter ünbernimmt dabei Takt für Takt die Abgleichung der Ergebnisse. Diese Strategie ermöglicht eine schnelle Korrektur von Fehlern ohne zeitaufwändige Wiederherstellungsverfahren (Recovery-Verfahren), so dass auch Echtzeitanwendungen unterstützt werden. Der *High-Performance*-Modus schöpft die volle Leistungsfähigkeit des Multiprocessorsystems aus. Zur Evaluation des Frameworks wurde eine neue Umgebung zur automatisierten Fehlerinjektion sowie eine neue Verifikationsplatform für Multiprozessoren entwickelt. Es wurden eine Vielzahl an Experimenten durchgeführt, die zur Bildung einer Formel verwendet wurden, welche bei gegebener Fehlerrate und Einsatzdauer die Anzahl an benötigten Prozessorkernen bestimmt. Darüber hinaus wurde basierend auf der Weibul Verteilung eine neu Methode zur Evaluierung der Lebensdauer entwickelte, die die Vorteile der Abschaltung der Kerne aufzeigt. Beispielsweise zeigt das Youngest-First Round-Robin (YFRR) Scheduling verglichen mit dem Standard-Round-Robin Verfahren eine Steigerung der Systemlebensdauer um 31%.

Acknowledgements

I would like to express my gratitude to the federal state of Brandenburg, Germany for granting me the scholarship in the frame of the ZUSYS (ZUverlässige SYSteme) group of the International Graduate School (IGS) at BTU Cottbus-Senftenberg, led by Prof. H. T. Vierhaus. Many many thanks to my mentor, Prof. Rolf Kraemer in the first place, a professor at the same university, and a head of the System Design department at the Institute for high-performance microelectronics IHP in Frankfurt (Oder), for recognizing my deep interest in microelectronics, for inviting me to apply for this position, and for his guidance throughout the studies. I'd also like to thank my previous mentor, Prof. Dr. Aristotel Tentov, a head of the Computer Technologies and Engineering Department at the Faculty of Electrical Engineering and Information Technologies (FEEIT) in Skopje, Macedonia, for enabling me to advance very fast, and for preparing me to undertake a work of this size. Together with my father, these three professors largely affected my scientific and engineering way of thinking, for which I'm grateful.

This work would not be possible without the help of many of my colleagues in IHP and FEEIT. Many thanks to Milos Krstic (IHP) for his inexhaustible support, the numerous suggestions and advice in both scientific, technical and organizational issues. Furthermore, I would like to thank Vladimir Petrovic, Steffen Zeidler, Oliver Schrape, Patryk Skoncej and Cirillo Maurizio (all from IHP) for the suggestions regarding many technical issues. Special thanks to Oliver Schrape for the enormous help regarding the layout, as well as to Irina Matthaei and Peter Dähnert (all from IHP) for the testing of my 8-core multiprocessor chip produced in IHP. Of course, a huge thanks to IHP for enabling the chip production! I would also like to thank Elena Hadzieva (FEEIT) for the very successful cooperation resulting in several scientific papers: Elena, thank you for proving formally that my intuition was right – scalable, programmable N-modular redundant voters could be easily constructed! These voters are essential for my thesis. Finally, many thanks to my brother Igor Simevski (FEEIT) for building a C compiler for my multiprocessor.

At the end, I would like to thank the members of my closest family, especially my wife Vesna, for her vicarious support, and our kids Natalija and Dushan for giving me huge amounts of positive energy needed to finish this work. I cannot find the words to express my gratitude to my mother and father, Lenka and Dushan. Mom and dad, thanks for everything!

I dedicate this work to my dear wife Vesna,
and our dear kids Natalija and Dushan.

Contents

List of Own Publications

[SH13] A. Simevski and E. Hadzieva. Software Implementation of Programmable NMR Voters. In *Electronics, Telecommunications, Automatics and Informatics (ETAI), 2013 XI international conference on*, September 2013.

[SHKK12] A. Simevski, E. Hadzieva, R. Kraemer, and M. Krstic. Scalable design of a programmable NMR voter with inputs' state descriptor and self-checking capability. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 182–189, June 2012.

[Sim13] A. Simevski. A simple and flexible 64/32-bit RISC core for embedded multiprocessors. In *Electronics, Telecommunications, Automatics and Informatics (ETAI), 2013 XI international conference on*, September 2013.

[SKK11a] A. Simevski, R. Kraemer, and M. Krstic. An Overview of Dependable Microprocessor Architectures - Pursuing the State-of-the-art. In *Electronics, Telecommunications, Automatics and Informatics (ETAI), 2011 X international conference on*, September 2011.

[SKK11b] A. Simevski, R. Kraemer, and M. Krstic. Low-complexity integrated circuit aging monitor. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 121 –125, April 2011.

[SKK12] A. Simevski, R. Kraemer, and M. Krstic. Platform for Automated HW/SW Co-verification, Testing and Simulation of Microprocessors. In *13th IEEE Latin American Test Workshop (LATW), 2012*, April 2012.

[SKK13a] A. Simevski, R. Kraemer, and M. Krstic. Automated integration of fault injection into the ASIC design flow. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*, pages 255–260, 2013.

[SKK13b] A. Simevski, R. Kraemer, and M. Krstic. Register-Transfer Level NMR System Generator. In *Zuverlssigkeit und Entwurf - 7. ITG/GI/GMM-Fachtagung*. VDE Verlag GmbH - Berlin - Offenbach, September 2013.

[SKK14a] A. Simevski, R. Kraemer, and M. Krstic. Increasing multiprocessor lifetime by Youngest-First Round-Robin core gating patterns. In *Adaptive*

Hardware and Systems (AHS), 2014 NASA/ESA Conference on, July 2014.

[SKK14b] A. Simevski, R. Kraemer, and M. Krstic. Investigating core-level N-modular redundancy in multiprocessors. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14) , 2014 IEEE 8th International Symposium on*, September 2014.

List of Tables

List of Figures

Chapter 1

Introduction

Semiconductor-based electronics enables complex systems that have enormous impact in virtually all aspects of human life. For instance, the world-wide mobile and telephone networks and the Internet are built using the products and services of the computing and communications industries, as well as other industries such as the space industry (e.g., for satellite interconnections between the nodes of these networks). Moreover, these networks are part of even greater systems, such as companies, cities, entire states, or international organizations. An interesting upside down pyramid that shows the pervasive use of semiconductors is presented in Fig. 1.1. Nowadays, one can certainly state that after the stone, bronze, and iron age, the mankind is currently in the electronic age, where the semiconductor-based electronics lies at the very base.

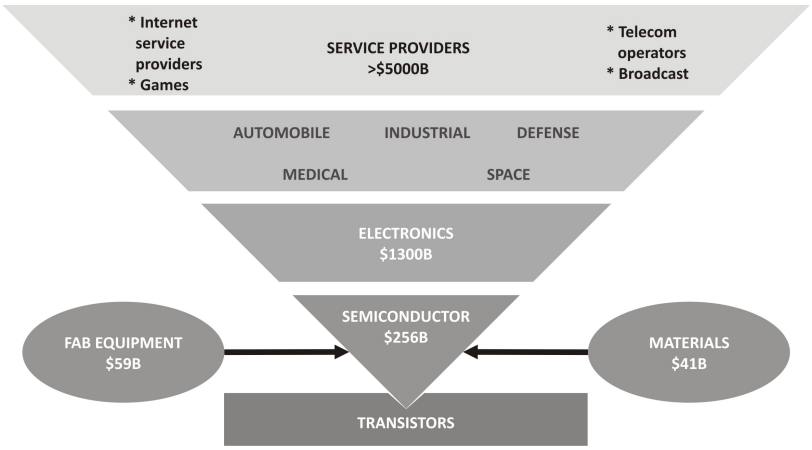


Figure 1.1: Semiconductor industry opens endless possibilities. (Adapted picture from a presentation by Frank Huang at the Green IT International Symposium 2008)

The question is what if the lowest part of the pyramid in Fig. 1.1 fails? Or, can one rely on it? If the semiconductor industry does not provide reliable systems, that are potentially build out of unreliable components (transistors or integrated circuits), all other systems built on top of it would not be reliable too. The importance of

reliability at this lowest level is clear. Fig. 1.1 also shows the profitable economic model, which aspect is frequently neglected by scientists. Each industry level has several times greater market (in billions of dollars) than the level below.

The term *reliability* is actually an attribute of a broader term – *dependability*. Dependability is one of the fundamental system properties besides *functionality*, *performance* and *cost*. The term performance is also a broad term that subsumes *speed of operation* and *power consumption*. Nevertheless, this term is widely used to denote only the speed of operation of the system, which is also the case in the thesis.

This dissertation deals with several aspects of multiprocessor's dependability. Section 1.1 gives the basic definitions that will be used throughout the thesis. The reliability failure mechanisms and how the technological downscaling affects them are explained in Sections 1.2 and 1.3, respectively. Section 1.4 shortly introduces the motivation and the proposed approach that is trying to address many challenges regarding multiprocessor dependability. At the end of this Chapter, Subsection 1.4.4 presents the organization of the thesis.

1.1 Dependability of systems – basic definitions

Many of the stated definitions in this Section are taken from Avizienis *et al.* [13]. They introduce a dependability tree (see Fig. 1.2) that clearly classifies the terms related to dependability.

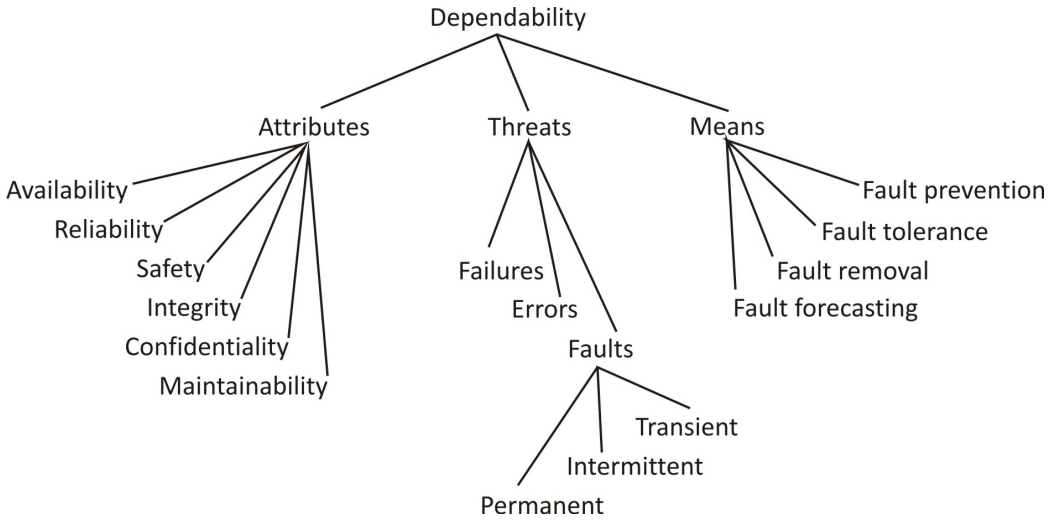


Figure 1.2: Dependability-related terms arranged in a tree

The **dependability of a system** is its ability to operate in a way that can justifiably be trusted. **Correct operation** is perceived when the system implements the system function that is described by its *specification*. Logically, **incorrect operation** is perceived when the system does not implement the system function.

1.1.1 Attributes of dependability

The attributes of dependability are:

- **Availability** – readiness for correct operation
- **Reliability** – continuity of correct operation
- **Safety** – absence of catastrophic consequences on users and environment
- **Confidentiality** – absence of unauthorized disclosure of information
- **Integrity** – absence of improper system state alterations
- **Maintainability** – ability to undergo repairs and modifications

Based on these attributes, dependable systems are also classified:

- **Ultra-reliable** – used in timing-critical control applications like in nuclear plants, or the avionic computers for unstable air-crafts (NASA), where the failure probability is less than 10^{-9} for a 10 hour mission [28].
- **Safety-critical** – where safety is the primary objective, e.g., in air-crafts, gas sensors, microwave applications, nuclear plants, biomedical implants.
- **Mission-critical** – e.g., space-crafts without crew.
- **Long-life** – where maintenance and repair is impossible, e.g., satellites.
- **Highly-available** – where downtime is expensive, e.g., telephone switching systems.

The main focus in the thesis are reliable, timing-critical, long-life multiprocessor architectures. Availability and maintainability are also investigated to some extent.

1.1.2 Threats to dependability

A **failure** occurs when the perceived operation of the system deviates from the correct operation. In other words, a failure is a transition from correct to incorrect operation. Failure may occur either because the system operation deviates from its specification, or the specification itself does not appropriately describe the function of the system.

An **error** is an incorrect part of the system state which may cause a failure. A failure is observed when an error reaches the output of the system. A **detected error** is signalled by the system through error message(s) or signal(s). A **latent error** is an error that is not detected by the system.

A **fault** is the adjudged or hypothesized cause of an error. An **active fault** produces an error, while a **dormant fault** does not produce an error. The term "fault" actually denotes an anomalous physical condition in the system, that could be caused

by various sources such as manufacturing problem, fatigue, external disturbance, design flaw... Section 1.2 shows a detailed classification of Integrated Circuit (IC) faults, as well as the most significant sources of IC faults i.e., failure mechanisms.

In other words, an error is the effect of an activation of a fault, i.e., manifestation of a fault. A failure is an over-all system effect or manifestation of an error. Fig. 1.3 presents this failure sequence.

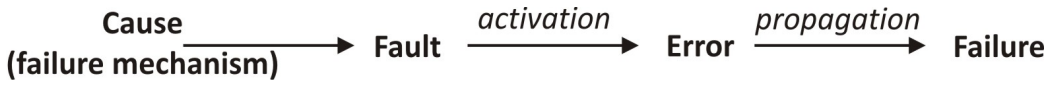


Figure 1.3: Failure sequence. (Adapted picture from [13].)

Regarding the persistence, faults are classified as permanent, intermittent or transient. **Permanent faults** occur due to manufacturing defects, early life (burn-in) stress, wear-out, etc. **Transient faults** are caused by external disturbances such as radiation, noise or electromagnetic interference. **Intermittent faults** or recurring faults are usually caused by marginal design parameters. They manifest timing problems, races, skew, etc., or signal integrity problems like crosstalk and ground bounce.

1.1.3 Means to achieve dependability

As Fig. 1.3 shows, if faults are not present, errors and failures will not occur in the system. Therefore, the means to achieve system dependability have the term "fault" at the beginning.

Fault prevention techniques aim at preventing introduction of faults in the system. These techniques include radiation hardening, shielding, following strict design rules, rigorous maintenance procedures, etc.

Fault tolerance is achieved by techniques that try to enable correct operation of the system in the presence of faults. Widely used approach to fault tolerance is error detection and recovery. A **recovery** procedure transforms an erroneous system state to an error-free system state. Both error detection and recovery could take place during normal system operation i.e., concurrently, or when the system operation is temporarily suspended i.e., preemptively.

The recovery procedures do fault and/or error handling. **Error handling** may be done by **rollback** to a previously known and saved, error-free state i.e., **checkpoint** or by **rollforward** to a new error-free state.

The **fault handling** procedure has four steps: fault diagnosis (identifying and locating errors), fault isolation (physical or logical exclusion of the faulty components), system reconfiguration (switching to spare components or reassigning tasks) and system reinitialization.

Fault masking is a form of concurrent recovery where error detection may not be done at all. The system simply relies on redundant components with the same functions. If one or more of the components have different states than the other

components that are majority, the error-free state is recognized in the majority components. Of course, this would function as long as there is a majority group of components with the same states.

Fault removal is done both during development of the system, and during its operational lifetime. Fault removal during development is performed by verification that the system adheres to certain properties that will enable correct operation. If this verification fails, diagnosis and correction of the design have to be done. On the other side, fault removal during operational lifetime is in the form of corrective or preventive maintenance. Corrective maintenance tries to remove the faults (causing errors) that are detected and reported by the system, while preventive maintenance tries to remove faults before they cause errors. Maintenance is different from fault handling since it is done by an agent which is external to the system. Fault handling is done by the system itself.

Fault forecasting is done by qualitative and quantitative evaluation of the system behaviour under activation of faults. Fault injection (see Subsection 1.1.4) is a widely used technique for evaluating system reliability and forecasting system behaviour in the presence of faults.

1.1.4 Measuring and evaluating dependability

Dependability calculations are heavily based on the probability theory [95]. Reliability as a function of time $R(t)$ is defined as the probability that the system will operate correctly under specified circumstances in a defined period of time. Expressed mathematically, $R(t) = P(TTF > t)$, where TTF is a continuous random variable that denotes the **time-to-failure**. It is assumed that $t \geq 0$ in all the equations in this Section. On the other side, $F(t) = P(TTF \leq t)$ shows the probability that the system will fail by time t, i.e., the cumulative distribution function (CDF) of failure. It is obvious that $P(TTF > t) + P(TTF \leq t) = R(t) + F(t) = 1$. Thus,

$$R(t) = 1 - F(t) = 1 - \int_{-\infty}^{t} f(s)ds = \int_{t}^{\infty} f(s)ds, \tag{1.1}$$

where $f(s)$ is the probability density function (PDF) of failure.

Empirical results show good matching to measurements if TTF is assumed to be exponentially distributed [108], in which case, the PDF of failure would be $f(t) = \lambda e^{-\lambda t}$ for $t \geq 0$. Thus,

$$R(t) = \int_{t}^{\infty} \lambda e^{-\lambda s} ds = e^{-\lambda t}. \tag{1.2}$$

where as shown below, the constant λ turns out to be the **failure rate**.

MTTF, MTTR, MTBF

According to the probability theory, $E[TTF]$ (the expectation, i.e., the mean value of TTF) is a weighted average of all possible values of TTF:

$$E[TTF] = \int_{-\infty}^{\infty} tf(t)dt. \tag{1.3}$$

$E[TTF]$ is a widely used metric for stating the reliability of systems which is commonly known as **Mean Time To Failure (MTTF)**. Replacing $f(t) = -\frac{d}{dt}R(t)$ (see Eq. 1.1) into Eq. 1.3 and adopting that the time is non-negative, $t \geq 0$,

$$MTTF = -\int_{0}^{\infty} tdR(t)dt = -tR(t)\Big|_{0}^{\infty} + \int_{0}^{\infty} R(t)dt. \tag{1.4}$$

Assuming that the system will inevitably fail, both limits of the first term in Eq. 1.4 are 0, ($\lim_{t\to\infty} tR(t) = 0$). Thus,

$$MTTF = \int_{0}^{\infty} R(t)dt \tag{1.5}$$

For exponential distribution of TTF (eq. 1.2), $MTTF = \int_{0}^{\infty} e^{-\lambda t}dt = 1/\lambda$.

Maintainability as a function of time $V(t)$ is defined as the probability that the failed system will be brought back to correct operation by time t. $V(t) = P(TTR \leq t)$, where TTR is the **time-to-repair**. The mean downtime, or the **Mean Time To Repair (MTTR)** is also a common measure, reflecting the maintainability of the system. It can be derived similarly as MTTF. For exponentially distributed TTR, the PDF is in the form $\mu e^{-\mu t}$. Thus, $MTTR = 1/\mu$.

Availability as a function of time $A(t)$ is defined as the probability that the system will operate correctly and implement its function in time t.

$$A(t) = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}, \tag{1.6}$$

where **MTBF** is the **Mean Time Between Failures**. MTBF is sometimes incorrectly used as MTTF, since practically $MTTR \ll MTTF$. In systems that are not repairable, MTTR and MTBF are not used at all, implying $A(t) = R(t)$. For repairable systems, $A(t) \geq R(t)$.

The bathtub curve, the failure rate and the hazard function

The bathtub curve is a widely used function for approximating the failure rate of complex systems [47]. The failure rate $\lambda(t)$ is the frequency with which the system

fails i.e., failures are observed. It is usually measured in FITs (Failures In Time), that is the number of failures in 10^9 hours of operation of the system. Fig. 1.4 shows the bathtub curve which is a superposition of three functions: the early life (infant mortality), the constant (random), and the wear-out failure rate. Equation 1.2 holds in the operational i.e., working time of the system where the failure rate is considered constant. ($\lambda(t) = \lambda = const.$)

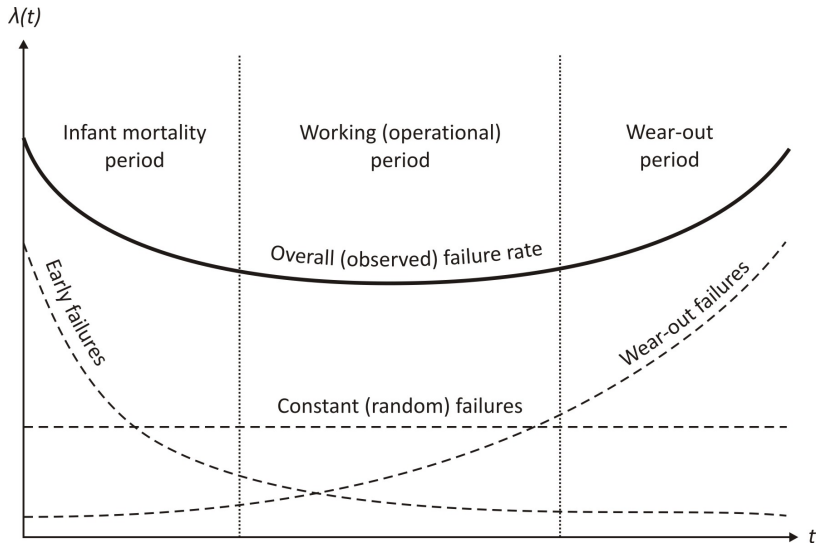


Figure 1.4: Bathtub curve representing failure rate

Expressed more precisely, the failure rate is defined as the probability that a failure occurs in a time interval $[t_1, t_2]$, given that no failures occurred prior t_1. This definition shows a clear relation between the failure rate $\lambda(t)$ and the reliability $R(t)$. Namely, the probability that a system fails in the interval $[t_1, t_2]$ is

$$\int_{t_1}^{t_2} f(t)dt = \int_{t_1}^{\infty} f(t)dt - \int_{t_2}^{\infty} f(t)dt = R(t_1) - R(t_2).$$

Thus, the failure rate is

$$\lambda(t) = \frac{R(t_1) - R(t_2)}{(t_2 - t_1)R(t_1)} = \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)}. \tag{1.7}$$

In the last part of eq. 1.7, the time interval $[t_1, t_2]$ is redefined as $[t, t + \Delta t]$. If the length of the interval approaches zero, the instantaneous failure rate, i.e., the hazard function $h(t)$ is obtained:

$$h(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)} = -\frac{dR(t)}{dt}\frac{1}{R(t)} = \frac{f(t)}{R(t)}. \tag{1.8}$$

For exponential distribution $h(t) = (\lambda e^{-\lambda t})/e^{-\lambda t} = \lambda$, that is, the hazard function is independent of time. In other words, the exponential distribution is memoryless. It is the sole continuous distribution possessing this property.

Fault injection

A **fault injection** procedure deliberately inserts faults in the system in order to evaluate the system behaviour, examine the error resilience, measure the failure rate, etc. Evaluating fault-tolerant mechanisms by fault injection is a common practice. It could be done in many different ways such as by simulation at various levels, emulation using FPGAs, pulsed-laser injection, heavy ion irradiation or electromagnetic interference.

When fault injection is done by simulation or emulation, **fault models** [128] are used to logically represent the types of faults that are to be injected. Although fault modeling is a concept that comes from integrated circuit testing where models like stuck-at, stuck-open or bridging are used to find out structural errors, it is also fundamental by simulated/emulated fault injection. Here, additional logical models like bit-flips or forcing a logical value in a signal may be used. A procedure for automated integration of fault injection into the ASIC design flow [SKK13a] is used to evaluate the fault-tolerant mechanisms of the proposed architectural framework. Several models (like stuck-at-0, stuck-at-1, bit-flip, force-0 or force-1) can be used for permanent, intermittent or transient fault modeling. Section 5.2 extensively elaborates the procedure.

1.2 IC reliability failure mechanisms

ICs produced in CMOS, Bipolar, BiCMOS, ECL, or any other technology (with CMOS being the most prevalent), fail over time. Although circuits produced in different technologies experience the same or similar failure mechanisms, there are also technology-specific failure mechanisms. For example, the aging effects observed in MOS transistors are different and far more severe than the ones observed in bipolar transistors [74]. This section discusses the reliability failure mechanisms of ICs, especially for CMOS ICs.

Fig. 1.5 shows an IC fault classification tree, and the most significant mechanisms that trigger each fault type. **Spatial faults** or **defects** are observable immediately after IC production and are fixed in time. They are related to the design and manufacturing process. The (non-)presence of defects determines the IC **quality**[1].

Defects stem from the process conditions and variability, the circuit layout, structure, geometry, environment, etc. They can be systematic or random.

[1] As said, *reliability* is the ability of the system to perform according to specifications under stated conditions and for a specified period of time T, for $0 \leq t \leq T$. On the other side, **quality** is the ability of the system to perform according to specifications under stated conditions at time $t = 0$, i.e., $Q = R(0)$.

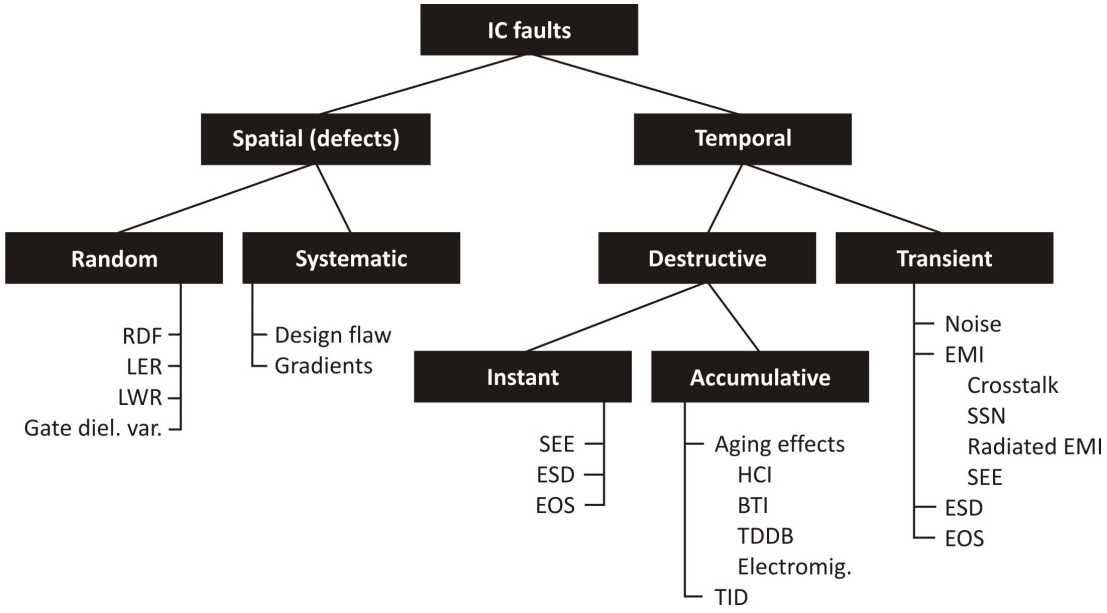


Figure 1.5: IC reliability failure mechanisms and fault classification. Fault classification tree is shown using white letters on black background. Failure mechanisms for each leaf are shown using black letters on white background. Some mechanisms like HCI, BTI and TDDB aging effects are specific only to CMOS ICs.

Systematic defects appear across large dimensions, e.g., entire die or wafer and are usually related both to design and to manufacturing process. For instance, the inability of the process to separate two wires that are put too close in the layout will lead to a systematic defect after production, resulting with bridged wires, possibly in each die and in each wafer. Gradients are another type of systematic defects that arise due to variation of certain physical parameters (e.g., temperature, pressure, oxide thickness) across an IC. Since these variations are mathematically expressed as 2D fields, their name comes from their mathematical treatment.

Random defects occur during the production process and can not be controlled or predicted. They include: Random Dopant Fluctuations (RDF) – fluctuations in quantity of dopant atoms; Line Edge Roughness (LER) and Line Width Roughness (LWR) stem from the sub-wavelength lithography; Gate Dielectric Variations stem from variations in the oxide thickness, introduction of fixed charges in the oxide or interface traps. Defects are actually permanent faults causing malfunctioning ICs, or ICs with degraded performance. E.g., material defects, cracks, oxide breakdown lead to a malfunctioning IC, while RDF for example, can cause parametric variations (like deviation of the threshold voltage or the drive current of transistors) that degrade the performance of the IC.

On the other side, **temporal faults** are not observable right after IC production, and are variable in time. The resilience to these faults determines the IC reliability. Temporal faults could be destructive or transient (non-destructive). They are related to the IC properties, operating and environmental conditions.

Destructive faults cause a permanent damage to the IC, leading to malfunctioning or degraded performance. Destructive faults could be instant or accumulative. **Instant faults** could be triggered by several mechanisms. For example, high-energy particle that hits the IC, could induce sufficient current to burn one or several transistors. ElectroStatic Disccharge (ESD) or Electrical OverStress (EOS) could cause dielectric breakdown due to the high voltage applied across the oxide. **Accumulative faults** come from mechanisms that continuously affect the circuit. The aging effects are such mechanisms. Another mechanism is the Total Ionizing Dose (TID), i.e., build-up of trapped charge in the oxide due to ionizing radiation. When active (not dormant), accumulative faults are usually intermittent at the beginning. Then, over time they become permanent. Since this thesis is related to long-life multiprocessor architectures, aging effects are of special interest. Therefore, Subsection 1.2.1 gives a more detailed introduction in this topic.

Transient faults, on the other hand, are not destructive and disturb circuit operation for a limited time period. Mainly two types of sources cause transient faults: noise and Electromagnetic Interference (EMI). Nevertheless, other sources like ESD and EOS may not be destructive and cause transient faults. Noise is defined as a random, unwanted deviation of a signal from its intended value. Noise is inherent to the circuit itself, i.e., it comes from the circuit. The production process largely determines the noise figure of circuits. EMI is defined as an influence of (source) signals on other, (victim) signals through a conductive, capacitive, magnetic or radiative coupling path. EMI also causes unwanted deviation of a signal from its intended value, but the source is external to the victim signals. The source signal could be natural (random) or man-made (deterministic). Natural EMI sources are for example, a thunderstorm lightning or cosmic noise. Man-made source signals could be accidental and unrelated to the victim circuit (e.g., switching on a power engine or microwave oven) or functional (e.g., interconnect crosstalk). The most frequent sources of EMI are the following. *On-chip crosstalk* can occur between two circuits, sub-circuits or elements of the same circuit. *Simultaneous Switching Noise (SSN)* is a special case of crosstalk, when circuits share the same power lines. This is commonly known as ground bounce, substrate noise or power-ground noise. The simultaneous switching of digital signals (tied to a clock signal) produces relatively large current spikes that are actually the source of SSN. *Radiated EMI* occurs when an unintentional transmitter of electromagnetic waves affects a victim circuit. These unintentional transmitters could be for example, mobile phones, power engines, microwave ovens, etc. At the end, *high-energy particles* such as alpha, beta or gamma rays ionize the semiconductor material, where the formed charge could change a signal value (transient fault) or, as said, could be destructive and damage the circuit.

Operating in hostile space environment (e.g., electronics in satellites) is largely investigated topic. Faults that originate from the ionizing radiation environment i.e., high-energy particles are called **Single Event Effects (SEE)**. Since this thesis is related to multiprocessor architectures intended for space applications, this topic is introduced in more details in Subsection 1.2.2.

1.2.1 Aging effects

Aging effects gradually degrade circuit performance and eventually cause permanent faults. Different mechanisms are considered as aging effects. Electromigration, for example, is an aging effect that affects ICs in any technology since it is related to the metal layers of the circuit. On the other hand, significant aging effects in CMOS ICs are related to degradation of the gate-oxide. These are Hot Carrier Injection (HCI), Time-Dependent Dielectric Breakdown (TDDB) and Bias Temperature Instability (BTI). Fig. 1.6 helps explaining gate-oxide aging effects.

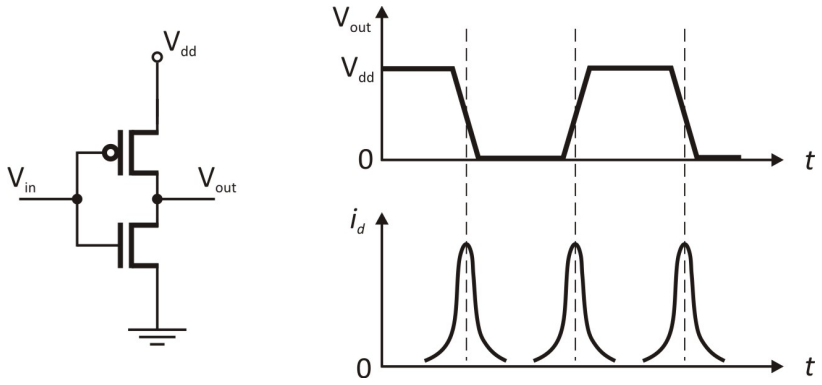


Figure 1.6: CMOS inverter. Significant current flows from Vdd to ground at each change of state.

Fig. 1.6 assumes operation of digital CMOS circuits. Analog CMOS circuits experience the same effects.

Hot carrier injection

There are several mechanisms that cause hot carriers i.e., electrons or holes with high energy to be injected into unallowed regions (gate and spacer oxide) of NMOS and PMOS transistors [74]. For example, when the gate voltage is (nearly) equal to the drain voltage of the NMOS transistor, the effect of injecting hot electrons into the gate-oxide near the drain is maximal. "Lucky" electrons gain sufficient energy (by not having collisions on their path from source to drain) to pass the barrier of the silicon – gate-oxide interface. Since holes are more heavier than electrons, this mechanism is more significant in NMOS transistors.

Another mechanism is impact ionization. When the drain voltage is high and the gate voltage is low, electron-hole pairs are created due to impact ionization of the channel current, again near the drain. The generated electrons and holes can accelerate in the electric field of the channel and potentially pass the silicon – gate-oxide barrier. This is known as avalanche multiplication and causes the most severe transistor degradation – a lot of carriers are injected into the gate-oxide at the same time.

In digital CMOS circuit operation, the channel current i_d (see Fig. 1.6) is responsible for HCI degradation. Both transistors conduct at each change of state which

shows a strong correlation between HCI aging and the frequency of operation of the circuit. That is, when the circuit is operating faster, it ages more due to HCI.

HCI causes shifts in the transistor parameters such as threshold voltage V_{th}, current factor β and output conductance g_0, leading to degraded transistor performance. Electrically, the input-to-output delay is increased due to HCI (the transistor gets slower).

HCI is temperature dependant [99]. At lower temperatures the mean free path for carriers is longer. Thus, carriers can gain higher energies. That means, the lower the temperature, the greater is the degradation due to HCI, which is especially adverse for space and aircraft applications.

To summarize, high performance (frequency) and low temperature increase HCI degradation.

Bias temperature instability

Negative BTI (NBTI) and Positive BTI (PBTI) are observed in PMOS and NMOS transistors, respectively. Nevertheless, PBTI is observed only in modern High-K Metal Gate (HKMG) technologies [40]. The impact of BTI is greater in smaller technology nodes, and becomes a serious issue. BTI also causes shifts in transistor parameters, especially in the threshold voltage V_{th}. Again, increasing V_{th} means an increase of the input-to-output delay, or slower transistor.

BTI is also temperature dependent (T in BTI). In this case it is the opposite of HCI, i.e., BTI degradation is increased with increase in temperature. The high electrical field at the gate-oxide combined with elevated temperature leads to the following process at the silicon – gate-oxide interface. Due to the different structure of silicon and gate-oxide, a thin interface region is formed. This region contains many dangling bonds (interface states) that trap and release carriers from the channel. Trapping carriers reduces carrier mobility and shifts the threshold voltage. During the manufacturing process, these dangling bonds are passivated by hydrogen so this process should not occur. Nevertheless, under negative (positive) bias and elevated temperature the passivized bonds could be broken again.

BTI has a specific property: the transistor degradation is recoverable to some extent immediately after reducing the stress voltage (due to releasing carriers by the dangling bonds). This complicates the modeling of the effect, but also measurements and lifetime estimations. V_{th} has to have permanent and recoverable part in modeling [39, 73]. The model of [39] includes the temperature dependency of BTI: $V_{th} \propto \exp(-\frac{E_a}{kT})$, where E_a is the activation energy and k is the Boltzman constant.

In contrast to HCI which causes asymmetric transistor degradation (carriers are injected near the drain), BTI causes homogeneous degradation and shows small dependency on transistor geometry.

Time-dependent dielectric breakdown

Dielectric materials do not conduct electrical current if the field across the material is lower than E_{max}, which depends on the material's properties, size and geometry. If

a larger field than E_{max} is applied, a Hard BreakDown (HBD) occurs, characterized by a local but large current flowing through the dielectric. EOS and ESD could cause HBD, as previously discussed.

The gate-oxide in a CMOS transistor is a dielectric with certain properties. In normal operating conditions the gate-oxide field is lower than E_{max}. Nevertheless, due to aging processes like HCI and BTI, the gate-oxide changes its properties until it fully breaks down and starts to conduct current in the range of milliamperes. As said, carriers are trapped into the gate-oxide due to HCI. With time, the larger and larger accumulation of carriers could form a conductive path through the gate-oxide, leading to dielectric breakdown. This is actually a time-dependent dielectric breakdown (TDDB).

HBD is the ultimate effect where the dielectric completely lost the insulating properties. Prior HBD, Soft BreakDown (SBD) may occur, with only partial degradation of insulating properties. SBD results in significant increase in the gate current noise [74]. Progressive BreakDown (PBD) could be also observed as a gradual increase of the gate current with time, prior HBD.

For modern nanometer technologies HBD could be a reliability threat only if the operating voltage of the transistor is higher than the nominal (normal) operating voltage [86].

Electromigration

The previous aging mechanisms are related to the gate-oxide of CMOS transistors. Electromigration, on the other side, is related to the metal wires, vias and contacts. Therefore, it is not only affecting ICs produced in CMOS technology, but also in any other technology.

As the name says, electromigration is an aging process where metal atoms (ions) migrate (are transported) due to high current densities and elevated temperature. The current electrons largely affect metal ions and transport them in the current direction. ICs (both with aluminum and copper interconnects) are very prone to electromigration [68]. The vias and contacts are especially affected. Although there are many attempts to reduce this effect (e.g., introducing copper in aluminum, or, tin in copper), electromigration is still a reliability threat in modern IC technologies [123].

Practically, moving metal ions from one place to another creates voids and hillocks in the interconnections. This in turn could cause changes in the metal resistance, as well as opens and shorts in the circuit.

1.2.2 Single event effects

Particles with high energy such as α and β particles, γ- and X-rays, heavy ions, protons and neutrons have negative impact on semiconductor electronic devices, starting from ionization to complete destruction. Environments with such high energy particles are referred to as ionizing radiation environments, while their effects on electronic devices are called **ionizing radiation effects**. The Van Allen belt around the planet Earth is such an environment, where many Low Earth Orbit (LEO) satellites reside.

Furthermore, solar flares periodically emit tons of protons and ions with high energy. Cosmic rays contain an abundance of particles with different energies (up to TeV). Thus, space electronics altogether is subjected to ionizing radiation. Lots of problems due to ionizing radiation in satellites or space missions have been reported so far. Moreover, ground applications in high energy physics and medicine (e.g., computed tomography, magnetic resonance and X-ray imaging) are also environments with high energy particles.

Ionizing radiation effects could be considered as a special type of electromagnetic interference (see classification in Fig. 1.5). They can be accumulative and instant. Regarding transitivity, they can be destructive or transient. Total Ionizing Dose (TID) is an accumulative radiation effect where charge is build-up in the oxide parts of semiconductors. The end effect is similar to the one caused by HCI or TDDB. Nevertheless, oxides thinner than 10 nm are much less sensitive to TID, i.e., modern technologies are immune (radiation hard) to TID effects [96]. On the other side, instant ionizing radiation effects (Single Event Effects) largely affect ICs in any technology.

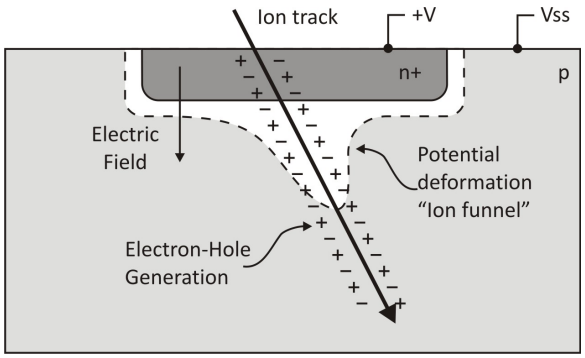


Figure 1.7: Ionizing high energy particle hitting a semiconductor

Fig. 1.7 shows a high energy particle hitting a semiconductor in the drain (or source) region. Impact ionization causes electron-hole pair generation in a dense track, both in silicon and oxide materials. The electrical field causes the generated carriers to drift and recombine. Nevertheless, many of the generated carrier pairs can survive. Thus, an amount of positive or negative charge could be collected at sensitive nodes in the semiconductor. If this charge is larger than a threshold value i.e., the *critical charge*, various effects (possibly destructive) could be observed in the circuit. This Subsection discusses these various single event effects that can arise [83].

The errors caused by SEEs could be classified as soft or hard. **Soft errors** are caused by transient SEEs and may be recovered by reset, by power cycle (off and back on), or simply by rewriting the memory element. On the other side, **hard errors** are non-recoverable since the fault causing it, is a destructive SEE. Fig. 1.8 shows the most significant SEEs observed in semiconductors, arranged and classified in a tree structure.

Single Event Upset (SEU) is a transient fault caused by an ionizing particle that induces a charge greater than the critical charge and modifies the electrical state

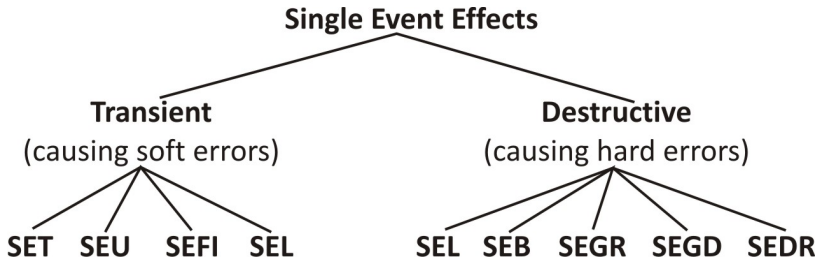


Figure 1.8: Classification of single event effects

of one or more memory elements. If more memory elements are affected, the effect is also commonly known as **Multiple Bit Upset (MBU)**. The error is produced when the value is read from the upset memory element(s).

Single Event Transient (SET) are transient voltage or current disturbances at the semiconductor nodes. In digital circuits, SETs could propagate to a memory element and practically become a SEU. If this is not the case, the SET would not trigger any error. On the other hand, in analog circuits SETs cause transient pulses in amplifiers, digital-to-analog or analog-to-digital converters etc, which could produce various types of problems.

Single Event Functional Interrupt (SEFI) is a complex effect that can occur in complex circuits where a SEU for example, disturbs a control register. This may induce a long series of errors and loss of functionality. Recovery is possible by reset, power cycle or rewriting the control register. SEFIs require special treatment in the system.

Single Event Latch-up (SEL) is an event that occurs when a parasitic PNPN structure in the semiconductor switches state. Besides by high energy particles, SELs could be triggered by electrical transients. Bulk CMOS technology is especially prone to this effect. A large increase of supply current is manifested, which could possibly destroy the semiconductor by overheating. Thus, SELs could be both transient and destructive. Recovery is possible only by a power cycle (assuming that the semiconductor is not already destroyed).

Single Event Burnout (SEB) and **Single Event Gate Rupture (SEGR)** are related to power semiconductors. These effects cause permanent damage to the devices. Before SEGR, usually **Single Event Gate Damage (SEGD)** is observed. The gate-oxide is eventually ruptured and starts to conduct. Although power MOSFETs are the most susceptible devices to these effects, similar effects (**Single Event Dielectric Rupture (SEDR)**) are observed in CMOS devices too [111].

1.3 Impacts of technology scaling

Since the emergence of ICs (1958), or more exactly the emergence of CMOS technology (1963), technology scaling has been (and still is) the main driver of the electronics industry. Scaling enables denser and faster integration of transistors on the silicon substrate, providing more and more electronic functions in a single chip with each

new generation. A famous citation of Jack Kilby, the inventor of the IC in 1958 at Texas Instruments, illustrates why scaling is the primary force behind the electronic industry: "What we didn't realize then was that the integrated circuit would reduce the cost of electronic functions by a factor of a million to one. Nothing had ever done that for anything before!"

In 1965, the co-founder of Intel, Gordon Moore, made a prediction based on a surprisingly little data. His original statement in [82] is: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. . . . no reason to believe it will not remain nearly constant for at least 10 years."

In 1975, Moore reformulated his statement which is known as the Moore's law: "The number of transistors will be doubled every 18th month." Practically, the semiconductor industry was defined. Based on the Moore's law, the ITRS (International Technology Roadmap for Semiconductors) constitutes a set of predictions for the future development of the semiconductor industry. Fig. 1.9 shows the confirmation of the Moore's law, using the available data from the past five decades.

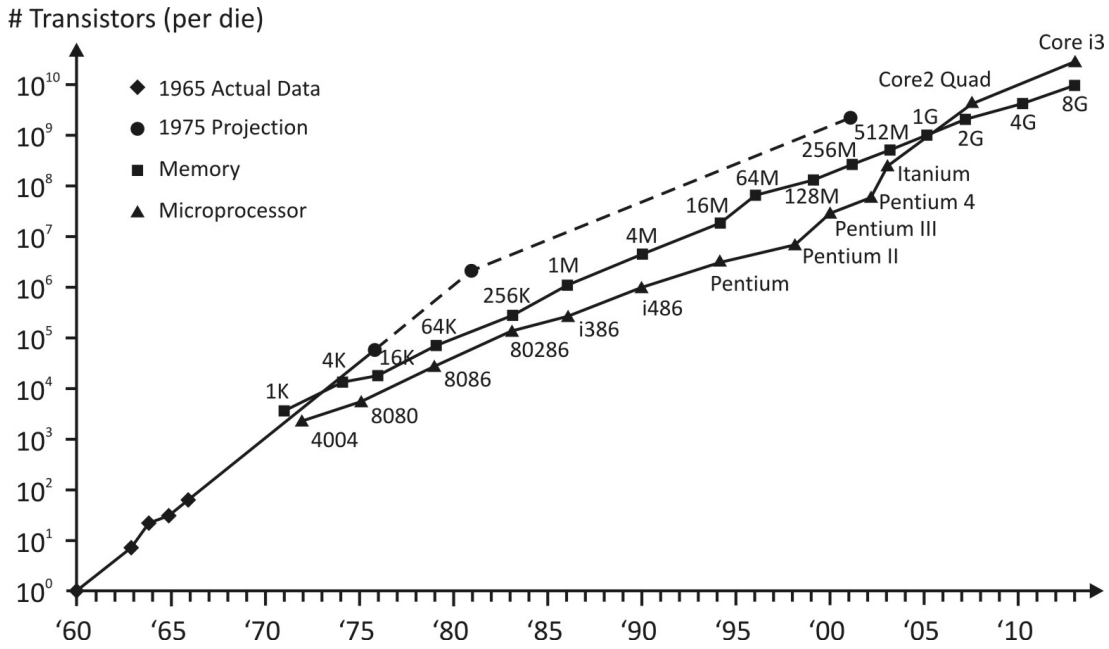


Figure 1.9: Confirmation of Moore's law. Moore had only the first 5 data samples when he made the famous statement in 1965. (Source: Intel)

Technology scaling brings lots of benefits: improves performance, increases the number of IC functions by increasing the transistor density, and reduces power consumption per transistor (or gate). The typical goals of technology scaling are the following [20].

- reducing gate delay by 30%, thus increasing the operating frequency of $\sim 43\%$
- doubling the transistor density

- reducing energy per transition of about 65%, thus saving 50% in power consumption at 43% increased frequency of operation

Despite the numerous benefits, technology scaling introduces many challenges and problems. Subsection 1.3.1 is devoted to the negative impacts of scaling on IC reliability. Besides lower reliability, ICs in each new generation have to deal with greater power consumption. Although 50% power is saved per transition in a transistor, the greater number of transistors, the increased operating frequency, as well as the growing die size, lead to an increased overall power consumption in each new generation. Eq. 1.9 shows that dynamic power consumption P_d in CMOS technology is proportional to the operating frequency f, the total IC capacitance (load plus internal) C, and the square of the supply voltage V_{dd}. Table 1.1 shows the trends using data from various generations of processors.

$$P_d \propto fCV_{dd}^2 \tag{1.9}$$

Excessive power consumption periodically appears as a problem. Until the mid 80's, NMOS and bipolar technology (which dissipate static power) were dominating. Furthermore, supply voltage was not scaling in order to keep simple interoperability between different technology nodes i.e., *constant-voltage scaling*. Afterwards, CMOS technology turned to be the single choice for digital VLSI circuits due to the negligible dissipation of static power. Scaling down the supply voltage was the next logical step, since dynamic power dissipation is proportional to the square of the supply voltage (see Eq. 1.9).

Currently, the power consumption is a problem again. Frequency scaling had to stop: as Eq. 1.9 shows, dynamic power consumption is directly proportional to the operating frequency. The data of the last decade shows that all high-end processors operate in the range of 1 to 4 GHz.[2] Moreover, die growth has also stopped [50] (see Table 1.1).

Already known "magic bullets" like **clock and power gating**, DVFS (Dynamic Voltage and Frequency Scaling), asynchronous design, etc. are used to reduce power consumption at the expense of another system property/ies (performance, cost and/or dependability). Reducing performance for example, always leads to lower power consumption. In other words, every design is a 2D point in a plane (see Fig. 1.10), that does not cross over the Pareto optimal curve [50]. Fig. 1.10 implies that a design trade-off has to be made, possibly at the optimal performance – power consumption point.

Of course, a larger design space (e.g., 3D or 4D) could be used in order to take into account other system properties.

1.3.1 Reliability trends

Technology downscaling has especially adverse impact on IC reliability. One could make such a statement only by looking at some simple facts. In a constant-voltage

[2] Over-clocking records go up to 9 GHz in newest technologies.

Table 1.1: Technology scaling trends. The largest part of the data in the table is collected from Intel. EL and DL denote technologies with enhanced and depletion load transistors, respectively. The power supply is given in Volts (V). Some of the older technologies require several supplies (separated with / in the table). In newer technologies, the operating voltage range is denoted by min-max allowed voltage. The power consumption is given in Watts (W). The maximal level of integrated (on-chip) cache as well as its size in Bytes (B) is also given, where L1:8K+8K for example, denote separate instruction and data cache at the L1 level, both with size 8KB. The litography resolution, the processor die area and the operating freqency are given in nm, mm^2 and MHz, respectively.

Year	Processor	Technology	nm	Tran.	MHz	V	W	Cache (B)	mm^2
1971	4004	EL PMOS	10000	2,3k	0,1	15	1	0	12
1972	8008	EL PMOS	10000	3,5k	0,5	5/-9	1	0	14
1974	8080	EL NMOS	6000	4,5k	2	5/-5/12	1,5	0	21
1976	8085	DL NMOS	3000	6,5k	3	5	1,5	0	20
1978	8086	DL NMOS	3000	29k	5	5	2,5	0	33
1979	8088	DL NMOS	3000	29k	5	5	2,5	0	33
1982	80286	CMOS	1500	134k	6	5	3,3	0	47
1985	i386	CMOS	1000	275k	16	5	2,3	0	104
1989	i486	CMOS	1000	1,2M	25	5	4,73	L1:8K	196
1993	Pentium	BiCMOS	800	3,1M	66	5	15,3	L1:8K+8K	294
1995	Pentium Pro	BiCMOS	600	5,5M	200	3,3	35	L1:8K+8K	306
1997	Pentium II	CMOS	250	7,5M	300	2	21,5	L1:16K+16K	203
1999	Pentium III	CMOS	180	28M	533	1,65	14	L2:256K	100
2000	Pentium 4	CMOS	180	42M	1500	1,59-1,75	57,8	L2:256K	217
2002	Itanium 2	CMOS	130	410M	1000	n/a	100	L3:3M	374
2002	Celeron M	CMOS	90	144M	1700	1-1,29	21	L2:1M	87
2004	Celeron D	CMOS	90	125M	2800	1,25-1,4	84	L2:256K	112
2006	Pentium D	HKMG	65	376M	3200	1,2-1,33	130	L2:4M	162
2007	Core 2 Quad	HKMG	65	582M	2660	0,85-1,5	105	L2:8M	286
2008	Core i7	HKMG	45	731M	2660	0,8-1,375	130	L3:8M	263
2010	Core i7	2D 2-gate	32	1170M	3330	0,8-1,375	130	L3:12M	239
2013	Core i3	3D 3-gate	22	1400M	3400	n/a	54	L3:3M	177

scaled technology, the transistors experience greater density of the electrical fields. The greater density of the field across the gate-oxide speeds up gate-oxide aging effects. Moreover, the smaller cross section of the metal conductors implies greater current density, thus worsening electromigration. On the other side, if voltage is scaled, the transistors have lower noise margins that make the IC more prone to electromagnetic interference and noise, while the conductivity of the metal conductors is lowered. A logical conclusion is that with technology downscaling, the lifetime of the IC gets shorter with increased number of FITs during the operational period. In other words, the bathtub curve in Fig. 1.4 gets narrower and goes up the y-axis.

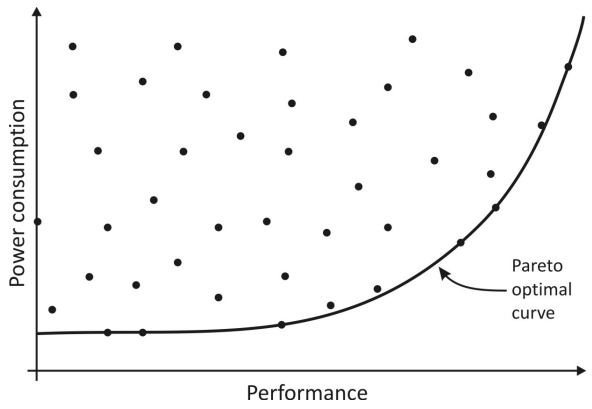


Figure 1.10: Trade-off between performance and power consumption. Each design is a 2D point. Designers should tend to locate designs at the Pareto optimal curve.

Aging effects trends

An elaborate description of the trends in aging effects is given in [74]. IC reliability issues were firstly observed at the beginning of the 1980s. Before that, the main reasons for system failures were corrosion, bonding problems (the purple plague) or ionic contamination, which are related to the IC packaging or the PCB (Printed Circuit Board). Scaling down the gate-oxide led to more and more visible degradation of IC performance due to the HCI effect, and eventually permanent faults due to TDDB.

Scaling-related problems with power consumption and reliability led to innovations in device structures and materials. A relatively great reduction in the gate leakage, the HCI and the TDDB effect was observed. Copper interconnects with low-k dielectrics were introduced to lower the RC-delays. Furthermore, high-k oxides and metal gates that enabled further scaling of the gate-oxide in CMOS devices led to the HKMG technology. Nevertheless, the new materials significantly amplified the so far negligible aging effects: NBTI and Electromigration. PBTI became equally important as NBTI, although it was not observed prior HKMG technology. In other words, the new materials introduced even greater reliability challenges. The newest FinFET technologies that use devices with 2 or 3 independent gates (multi-gate devices) also use these materials and experience the same effects.

Quantitative evaluation of lifetime reliability based on industrial-strength models for different technology nodes is given in [107], where the authors evaluate several aging effects in a POWER4-like processor implemented in 180, 130, 90 and 65 nm technology. The average failure rate goes from 2000 to 4000 FITs for the 180 nm technology, up to 18000 FITs for the 65 nm technology.

Soft error trends

Besides lifetime reliability issues, ICs in scaled technologies are becoming more prone to electromagnetic interference and noise, as mentioned. Of special interest are the

single event effects causing soft errors. Soft errors generate higher failure rate than all other reliability failure mechanisms combined [75]. A few quotations follow, illustrating the soft error challenges more clearly:

"Cosmic rays are almost impossible to stop. They'll go through 5 feet of concrete without any trouble, and cause a bit to flip." (Lange, IBM)

"In 0,13-micron technology we're seeing some memory technology with error rates of 10.000 or 100.000 FITs per megabit. This brings the frequency of error in a single device down to weeks or months." (Eric-Jones, MoSys)

"A system with a 1 GB of RAM can expect an error every two weeks; a hypothetical terabyte system would experience a soft error every few minutes." (Tezzaron Semiconductor)

As technology scales down, there is an exponential increase in the **Soft Error Rate (SER)**[3], caused by the most common source of single event upsets i.e., low-energy alpha particles [75]. The impact of the new technologies on SER is investigated in [29] using neutron beam irradiation of processors in different technology nodes. The results are shown in Table 1.2.

Table 1.2: SER by microprocessors in different technology nodes. The SER per bit decreases as technology scales from 250 down to 65 nm technology. Nevertheless, a reversal of this trend is observed at the 40 nm technology. (Source: [29])

Tech. (nm)	Relative SER (FITs/kbit)	Mbits/μp	Relative SER per μp (KFITs)
250	3,2	1,52	5,0
180	3,0	1,52	4,3
130	2,4	3,28	7,9
90	1,0	33,6	33,6
65	0,7	44,3	30,5
40	0,94	71,0	67,0

Table 1.2 also implies a tendency of increase of the SER/microprocessor as technologies scale down, simply due to the larger number of Mbits/microprocessor i.e., the greater complexity and silicon area enabled by the new technology.

An interesting trend is that the larger available silicon area of each new generation is occupied by memory [69]. Fig. 1.11 shows a 2007 prediction of memory and logic occupancy in ASICs, using the previously available data. Fig. 1.11 further suggests a logical assumption that on-chip memories are (and will be) responsible for the largest part of failures in the ICs.

1.3.2 Architectural trends – multiprocessing

The most widely accepted and used machine programming paradigm today is based on series of instructions (**programs**) that optionally use data operands in order to

[3]The SER is simply the number of FITs caused by soft errors.

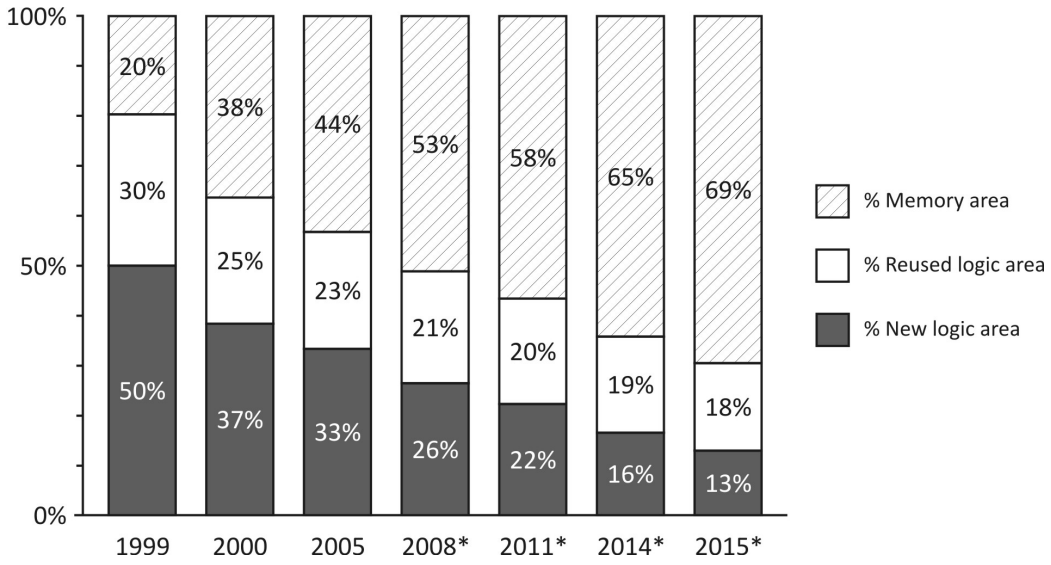


Figure 1.11: Memory and logic area trends in ASICs. Prediction from 2007. (Source: Semico Research Corp.) Actually, ITRS 2007 [56] predicts that memory area will occupy 94% of the total chip area by 2014.

perform operations like data or control transfers, arithmetic or logic computations. Based on the limited set of instructions, one could build Finite State Machines (FSM) that are able to process the instruction series. These FSMs are commonly known as **processors**, or, when implemented in microelectronic technology – **microprocessors**.[4]

Basically, speeding up execution of a program could be done in two ways. One is to increase the operating frequency of the processor, which is not an easy task mainly due to the excessive power consumption, as previously discussed. The second way is to exploit the available Instruction-Level Parallelism (ILP) in the program. That is, try to execute two or more instructions in parallel. The main obstacles of this approach are the data and control interdependencies. For example, if the results of one instruction are used as data in the next one, parallel execution of these two instructions is not possible. The contemporary microprocessor architectures have reached the point of nearly full exploitation of the possible ILP [44], thus coming to another wall. Other ways of further increasing performance have to be found.

The next logical step, and current trend is **multiprocessing** which is actually a known concept for decades. One or more programs are divided into separate **tasks** and scheduled for execution by one or more **Processing Elements (PEs)**.[5] The term **process** denotes the part of the program which is being executed by the PE.

[4]The terms processor and microprocessor are interchangeably used in the thesis, denoting processors implemented in micro- and nanoelectronic technology. The term nanoprocessor is rarely used in the literature, and not used here. It is also possible that the term processor is occasionally used to denote a multiprocessor (or, multi-core processor) realized in a single IC.

[5]The PE is more commonly called the **core**, although this term has sometimes different meanings. Thus, **multi-core** processors or systems are also multiprocessors.

Each PE in the system executes a time-multiplexed series of processes. A **context-switch** is a procedure when the processor temporarily stops the execution of the current process, and starts executing the next process scheduled for execution. This **scheduling** could be done in different ways in order to guarantee fairness between the programs, achieve efficient execution, give priority to some programs, retain predictability of execution, etc. A system with two or more PEs is called a **multiprocessing system**, or, **multiprocessor**. The PEs in the multiprocessor could simultaneously (i.e., in parallel) execute processes, thus speeding up operation. Of course, the interdependent tasks have to be executed sequentially, which here too, limits performance. Determining which PE will execute a given task is called **task mapping**. Another design challenge, and main differentiating characteristic of multiprocessing systems is the **interconnection network** of PEs and memory modules.

The term **thread** denotes a lightweight process that can not manipulate with the complete resources of the PE (especially with system resources), opposed to a "classical" process. One PE could be built to support execution of two or more threads simultaneously, which is called **multithreading**.

Multiprocessors are nowadays widely used in all computing segments: desktop, server and embedded. They present a great processing power to programs which could be divided into smaller procedures that are processed in parallel, e.g., scientific applications with intensive computations, or, applications with large number of independent processes, e.g., web-servers serving thousands of clients simultaneously. In the embedded domain, image processing or other computationally-intensive applications regularly opt multiprocessors. In the desktop domain, usually greater graphic processing power is required (e.g., for entertainment and games); Graphic Processing Units (GPU) deal with large amounts of data in parallel; besides that, the operating systems make use of the greater number of PEs in order to speed up critical operations.

1.4 Thesis proposal

The dependability challenges and trends, as well as the architectural trends driven by technology scaling are the basis of this work. To recap, technology downscaling brings lots of challenges regarding IC reliability: resilience to aging effects and failure mechanisms like single event effects is lowering! Furthermore, excessive power consumption is a problem again! This Section summarizes the motivation behind the thesis and shortly outlines the proposed solution.

1.4.1 Motivation

Logical and empirical conclusions show (see Section 1.3) that the best way to save energy and defy aging is to do less work i.e., lower performance, implying that the same techniques for reducing power-consumption could be used for reducing aging-effects, and vice versa. Although emerging technologies show significant improvements in power consumption (see Table 1.1), this is not the case with aging effects.

On the other side, the soft error rate is increasing. ICs are becoming more prone to electromagnetic interference and single event effects. More elaborate fault-tolerant mechanisms must be used to keep low error rates. As will be shown in Chapter 2, increasing fault tolerance could be done by reducing performance, by increasing the cost of the system (e.g., larger IC area), or, both. In other words, there are different trade-offs between aging, fault tolerance, performance, power consumption and cost of the system. Furthermore, Section 2.3 presents solutions based on the strategy used in this work: **dynamic adaptation to application requirements**.

That is, the main thesis here is that these challenges and trade-offs could be addressed by parallel operation, specialized functional units, application-level optimization and **intelligent adaptive control!** This is basically powered by the fact that most applications, especially in the embedded domain, dynamically change their requirements regarding performance (with that energy) and fault tolerance. Consider Example 1.

Example 1. *Satellites are designed to stay operational for ten or more years (long-life systems) with limited resources of power, mainly solar. Their environment is extremely hostile: rich with high-energy particles that disturb circuit operation, introduce permanent and transient faults, and accelerate aging effects. Maintenance by human intervention is not possible.*

Let a multiprocessor-based, on-board computer in an earth observation satellite is involved in several tasks concurrently. High computing performance is needed for the observation task i.e., image processing. Here, several erroneous image bits are not mission-critical and can be accepted, meaning that the expectations of reliability in these computations are not very high. On the other side, critical procedures like change (or control) of the satellite's orbit have to be executed with very high reliability i.e., in a fault-tolerant manner.

In situations where the multiprocessor activity is low (e.g., waiting for an interrupt for a new observation task), entering a low-power mode which could also extend the total lifetime of the system, is desirable. As explained, the more stress is caused by running the multiprocessor at full speed continuously, the shorter the lifetime.

The scenarios from Example 1 call for a high-performance, highly-reliable, long-life and low-power system. It is impossible to have some combinations of these properties in the same time and in full extent, due to the trade-offs discussed previously. Nonetheless, they are not actually required at once, e.g., image processing demands high performance but not high reliability. In the case of orbit control, exactly the opposite properties are required. In other words, application requirements are dynamically changing. An adaptive and flexible system, capable to adjust itself to the changing requirements would be a possible solution.

1.4.2 Proposed architectural framework

This work investigates such a flexible and scalable multiprocessor framework that besides dynamically adapting the multiprocessor to the application requirements regarding fault tolerance and performance, at the same time tries to prolong system

lifetime and save energy as much as possible. Several operating modes are considered, three of which are basic:

- de-stress mode,
- fault-tolerant mode,
- high-performance mode.

Dynamically switching these modes on a request by the operating system or other application would enable dynamical adaptation.

In **de-stress mode**, a minimum required number of multiprocessor cores are active (execute instructions), while all others are inactive (disconnected from the power supply or from the clock). The number of active cores should be determined according to the application requirements. Using core gating patterns like round-robin, the work of the currently active cores could be transferred to inactive cores (which become active), while the active ones are power- or clock-gated (thus becoming inactive). Such a takeover of the workload could be done repetitively in time by different cores. This systematic de-stress is supposed to increase the multiprocessor's lifetime: as said, aging is reduced or eliminated if the circuit is less active or not powered. Besides de-stressing, this mode also enables power-saving.

Fault-tolerant mode, on the other side, increases multiprocessor fault tolerance. $N > 1$ tightly synchronized cores are assigned to execute the same task simultaneously. An NMR voter determines the final core outputs at each clock cycle, thus forming a core-level NMR (N-Modular Redundant) system. The number of cores N could be assigned statically or dynamically. In static assignment, N remains fixed during the execution of the fault-tolerant task. In dynamic assignment, N could be changed during task execution. For example, the task may be initially executed by two cores forming a DMR (Dual-Modular Redundant) system. As long as the outputs of the cores match, an error-free operation is assumed. On a mismatch, new cores are dynamically assigned and the operation is restarted in order to recover from errors. Now, $N > 2$ cores synchronously execute the (restarted) task. For low fault rates, three cores forming a TMR (Triple-Modular Redundant) would be sufficient. This dynamic core assignment actually resembles a form of NMR On-Demand (NMROD) system (see Subsection 2.1.3).

At last, **high-performance mode** enables high-performance operation. When operating in this mode, the multiprocessor differs by nothing compared to a common multiprocessor.

The multiprocessor framework (or simply framework in further text) consists of

- **framework controllers** – hardware part of the framework that couples/decouples the modules to/from the power supply or clock in de-stress mode and forms NMR systems in fault-tolerant mode; the appropriate voters are part of the controllers;

- **framework middleware** – software that "drives" the framework controllers on one hand, and offers the framework services to the application layer on the other hand, hiding the hardware details.

The application layer (operating system, or other applications) call the middleware routines that program the framework controllers or read their status. Fig. 1.12 shows the layering and encapsulation of the system.

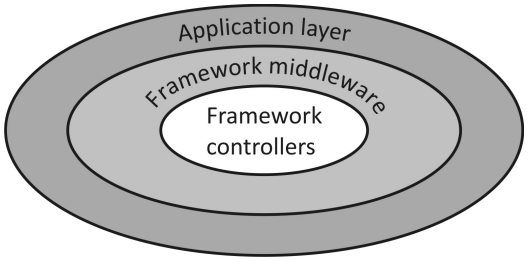


Figure 1.12: Layering and encapsulation

Fig. 1.13 presents a general multiprocessor system based on the proposed framework. In further text, it is simply called a **framed multiprocessor**. Each of the K framework groups are controlled by a framework controller. In a group i with P_i identical modules, NMR systems with $1 < N \leq P_i$ could be dynamically formed. The modules are connected by an interconnection network which should enable scalability and fault-tolerant (e.g., redundant) links.

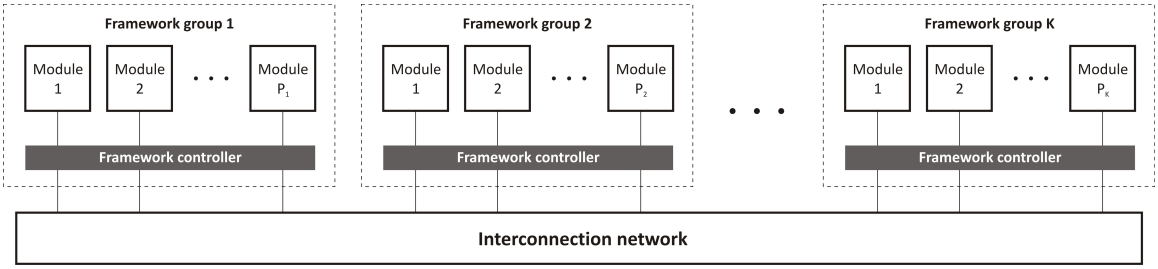


Figure 1.13: Framed multiprocessor – general architecture

The modules could be PEs, cores with or without caches, memory modules or even entire subsystems consisting of cores and memory modules.

Example 2. *A framed multiprocessor consists of two framework groups. The modules of the first group are three processors, while the modules of the second group are three memory modules (see Fig. 1.14).*

The system could be configured to operate in de-stress mode with one processor and one memory module, to save power and reduce aging. When fault-tolerant operation is required, the three processors could be arranged in a TMR system, where the output is chosen by a voter; in the memory group, only one memory module could be active, or, alternatively, the three memory modules could be also arranged in a TMR system. For high-performance, the system could be configured as a multiprocessor with three independent PEs and three independent memory banks.

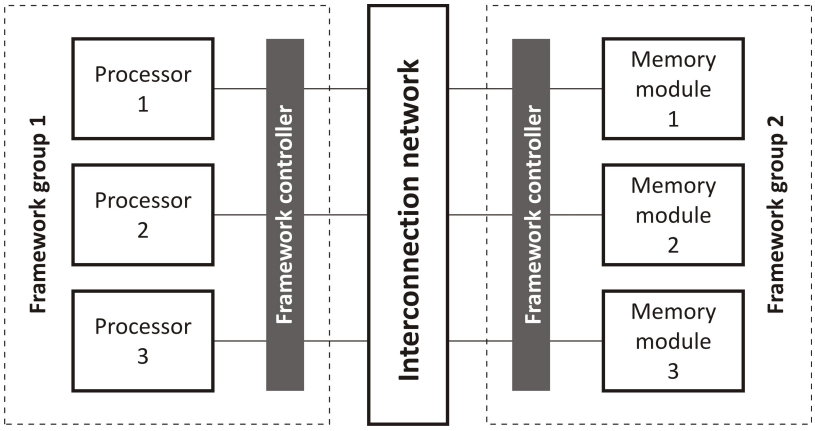


Figure 1.14: A specific example of a framed multiprocessor

1.4.3 Objectives

To sum up, the proposed multiprocessor framework could improve lifetime and fault tolerance, but also keep high-performance and lower power consumption by dynamically adapting the multiprocessor to the application requirements. In particular, the following benefits are expected.

- Meet performance and fault tolerance requirements of applications at the minimum possible rate of aging and power consumption; de-stress mode of operation inactivates modules that are not needed and systematically de-stresses the active modules using a de-stress pattern; de-stressing not only reduces aging effects (improves lifetime) but also reduces power consumption.

- Provide fault tolerance for timing-critical applications; multiprocessors are very suitable platforms for software-based fault-tolerant mechanisms; however, these mechanisms are often inappropriate for applications where lengthy error handling is not an option; the fault-tolerant operating mode is based on a coarse-grained NMR grouping of modules which instantly masks faults and enables uninterrupted operation; using a very flexible, programmable NMR voters, the framework could be instructed to build almost any NMR scheme, both static and dynamic (e.g., NMROD).

- Provide a high-performance mode that makes the framed multiprocessor operate as any common multiprocessor; however, the framework may employ lifetime-aware task mapping and scheduling using the information supplied by the embedded aging monitors; furthermore, the power- and clock-gating facility could be also used to inactivate unneeded modules in this mode.

- Introduce simple approach of building the system, where already verified processor cores could be used as building blocks, thus leveraging both hardware and software design.

- Provide scalability using a scalable network for module interconnection; the network should also enable redundant connections and simple routing.

1.4.4 Thesis organization

Chapter 1 gives the basic definitions and terms in the field of system's dependability; it elaborates the trends in micro- and nanoelectronics (i.e., technology downscaling trends); the motivation of the thesis is highlighted, and the proposed solution is outlined.

Chapter 2 presents an in-depth overview of the state-of-the-art in the field, especially emphasizing the solutions of the presented challenges and problems in Chapter 1; moreover, proposed solutions are appropriately classified according to some criteria (e.g., type of redundancy); a special Section is devoted to the contribution of this work to the field of investigation, that is, the progress that this work brings beyond the state-of-the-art.

Chapter 3 elaborates the proposed architectural framework at a general level; the three basic modes of operation (de-stress, fault-tolerant and high-performance) are explained thoroughly; hardware and software requirements are outlined for each mode; the Chapter ends with a section devoted to the scalability of the concept.

Chapter 4 goes deeply into the implementation details of both hardware and software part of the proposed framework; the three layers of the system, i.e., framework controllers, framework middleware and application layer (see Fig. 1.12) are explained in separate Sections; a special Section presents the design method and evaluation of one of the most critical components of the framework controllers – the programmable NMR voters; at the end, Section 4.5 presents an 8-core framed multiprocessor implemented in 130 nm technology which is used to evaluate the framework.

Chapter 5 explains the verification platform and methods; the novel, state-of-the-art procedure for automated integration of fault injection into the ASIC design flow is presented; this procedure is integrated into a broader environment for multiprocessor verification, which is also explained in details.

Chapter 6 is devoted to a thorough evaluation of the proposed concept; all experiments, theoretical evaluations and results are presented; the impacts on aging effects (lifetime), fault tolerance, performance and power consumption are assessed.

Chapter 7 sums up the most important conclusions; it discusses whether the proposed solutions meet the challenges claimed in Chapter 1, and whether the objectives are met; the planned future work is also outlined.

At the end, two appendices further discuss some practical aspects of this work.

Implementation details of the RISC core used in the novel multiprocessor implementation are presented in Appendix A.

Appendix B shows a table of all framework middleware procedures with short descriptions.

Note that throughout this writing, the own papers are cited in "alpha" style (first letters of authors' surnames plus publication year), while other references are cited using the "plain" style (ordinary numbers). The list of own publications is found on page xvii, while the bibliography is on page 161.

Chapter 2

Related work

The increasing rate of faults in ICs demands permanent research of innovative solutions that increase fault tolerance and extend systems' lifetime. There is an intensive search for solutions at all levels: introduction of new materials, improvements in the IC fabrication processes, development of special (e.g., radiation hardened) layout libraries, fault-tolerant circuit and system (architectural) design, etc. This Chapter makes an in-depth overview of circuit and system level techniques for improving fault tolerance and lifetime in digital circuits, especially in processors and multiprocessors. Techniques for reducing power consumption are also reviewed. An overview of dependable microprocessor architectures which is a part of this work is presented in [SKK11a].

Section 2.1 presents mechanisms for improving fault tolerance. Reducing aging and power consumption is reviewed in Section 2.2. Section 2.3 is devoted to techniques for adaptation to application requirements regarding fault tolerance, lifetime, performance and energy. The advantages and disadvantages of each reviewed solution (or group of solutions) are highlighted in all Sections. If applicable, a comparison to the thesis proposal is also made. At the end, the progress that this work makes beyond the state-of-the-art is presented in Section 2.4.

2.1 Increasing fault tolerance

Digital circuits inherently posses fault-tolerant characteristics. E.g., there is no impact on circuit operation if there is a bit-flip on a line that is logically "and-ed" with 0. Furthermore, if a transient disturbance (e.g., voltage or current spike) occurs for a short time interval (compared to the clock period), and does not enter the latching window of the appropriate flip-flop, no error occurs. In large designs, there is another phenomena that may mask faults: all functions are never used simultaneously. For instance, a transient fault in the division logic of the processor during a non-division instruction will not produce an error.

It is also worth mentioning that techniques used for testing, like Built-In-Self-Test (BIST) or scan-chains, whose original purpose is quality assurance, can be used for fault diagnosis too. For example, the dual-core UltraSparc processor has a scan

lockstep mode in which the scan chains in both cores receive the same inputs. If a mismatch in any bit in any cycle is detected, it is reported by an output pin.

Nevertheless, practice shows that these inherent mechanisms are not enough. In order to significantly reduce the system's FITs one should implement fault-tolerant mechanisms. All solutions for improving fault tolerance contain some form of redundancy. Broadly speaking, fault-tolerant mechanisms are classified into three categories based on the type of redundancy they introduce. That is, information, time and space redundancy. Nevertheless, complex systems may use or combine several mechanisms of different types.

2.1.1 Information redundancy

Digital data could be represented by more bits than necessary, thus providing a way to detect and correct errors. This systematic addition of redundant bits to the data bits, and forming sets of allowed combinations of values of all bits (data + redundant) is called **coding**. If the purpose is error detection and/or correction, it is called **error control coding**, or, channel coding in telecommunications.[1] There are lots of error detection/correction codes for different classes of errors, that use different amounts of redundancy. The complexity of design, as well as the ease of coding and decoding is also different.

The schemes presented in this Subsection are based on error detecting and correcting codes. They can detect both permanent and temporary faults using relatively small hardware overhead, at the expense of added complexity in design.

Error control coding

An error control coder takes a k-bit data on the input side and produces an n-bit code, where $n > k$ (see Fig. 2.1).

Figure 2.1: Error control coder used for error detection/correction ($n > k$)

Forward Error Correction (FEC) codes, commonly also known as ECC (Error Correction Codes), are principally divided as block and convolutional codes. **Block codes** encode data in blocks and are the largest family of error correcting codes. If the block has n bits, 2^n different blocks could be formed, partitioned into codewords and non-codewords. Thus, an error could be detected only if the codeword is transformed into a non-codeword.

[1] Besides error detection/correction, coding may be used for other purposes like data compression (source coding) or cryptography. In telecommunications and networking, another goal of coding is to improve transmission speed. Actually, network coding is used to improve throughput, efficiency and scalability.

A separable block code, denoted as (n, k), has k data bits that directly represent the message, while the rest $(n - k)$ bits are check bits. The advantage of separable block codes is that the message could be extracted right away, without decoding. The rate of a block code is defined as $R = k/n$. Lower rate usually means greater capabilities for error detection and correction but also greater complexity.

Example 3. *Single bit parity code* $(n, n-1)$ *is used for error detection only. The rate* $R = (n-1)/n$*. The check bit is a XOR function of the message bits. E.g., the message 0101 encoded by a* $(5, 4)$ *code would be 01010, where the added zero is the check bit, computed as* $0 \oplus 1 \oplus 0 \oplus 1$*.*

On the other side, a non-separable block code does not separate the check bits from the data bits i.e., decoding is needed.

Example 4. *Each codeword in the one-hot code has a single 1. E.g., a 4-bit one hot code has the codewords 0001, 0010, 0100 and 1000. All other combinations are non-codewords. This code is non-separable because the message is not contained in the block and has to be decoded.*

A special class of block codes are the *linear block codes* which have the property that for any two codewords C_i and C_j, the modulo-2 sum (exclusive or operation) $C_i \oplus C_j$ is also a codeword. Thus, the single bit parity code is linear, while the one-hot code from Example 4 is not linear.

All linear block codes have a $\mathbf{G}_{k \times n}$ generator matrix, and a $\mathbf{H}_{(n-k) \times n}$ parity check matrix. The generator matrix is used to construct the n-bit codeword from the k-bit data (see Fig. 2.1) such that $C = m\mathbf{G}$, where C is the codeword, and m is the message (the data). On the other hand, $C\mathbf{H}^T = 0$ (0 is the zero matrix), for any n-bit codeword C. Similarly, $C\mathbf{H}^T \neq 0$ if C is not a codeword. Furthermore, $\mathbf{G}\mathbf{H}^T = 0$. Thus, an error is detected by a simple check if the product of the codeword and the parity check matrix is not zero. Example 5 illustrates these matrices on the Hamming $(7, 3)$ code. This code has the possibility to pinpoint the bit location of the eventual (single) error in the examined vector v, by using the syndrome $s = \mathbf{H}v^T$. If $s = 0$, v is actually a codeword and no error is present. Thus, simple bit correction follows by inverting the bit on position s, if $s \neq 0$.

The *hamming distance* between two codewords is defined as the number of bits in which they differ. The *codeword weight* is defined as the number of ones that the codeword contains. For instance, the codewords 0110 and 0101 have a hamming distance and codeword weight of 2. The minimum hamming distance between any two codewords in the code shows the hamming distance of the code itself. A code with distance d can detect $d - 1$ and correct up to $\lfloor (d-1)/2 \rfloor$ errors in a block.

The Hamming codes are actually a family of linear block codes generalized from the original Hamming $(7, 4)$ code [41]. They can detect up to two bit errors or correct one bit error. Therefore, they are usually called SEC-DED (Single Error Correction – Double Error Detection) codes. The $(7, 3)$ Hamming code is only SEC. Hamming also gives a very simple procedure for constructing SEC code in the same paper [41]: any matrix with all distinct and non-zero columns is a $\mathbf{H}$ matrix of a SEC code. Example 5 illustrates error location and correction by a $(7, 3)$ SEC Hamming code.

Example 5. *Let each column in a* $\mathbf{H}_{7\times3}$ *matrix be equal to the binary representation of the column number. Let the first column start from 1. Thus, all columns will be non-zero and different to each other. Now, let the intended codeword* $C = 0110011$ *has a single bit error at the beginning, changing it to* $v = 1110011$*. The syndrome*

$$s = v\mathbf{H}^T = [1110011] \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = [001]$$

is equal to the first column, or, in other words pinpoints the erroneous first bit of v*. Simple inversion of this bit will correct the error. If* $s = [000]$*, no error is detected, and* v *is a codeword.*

Each Hamming code has an equivalent *cyclic code*. In fact, the largest class of linear block codes is "polynomially generated", or, cyclic. Their simplicity of implementation (shift registers with feedback connections), the rich algebraic structure and extremely concise specifications render out of use the non-cyclic codes [37]. They are called cyclic because each cyclic shift of a codeword is also a codeword. All codewords are actually shifts of one another, and can be represented and derived using generator polynomials $g(x)$. The Cyclic Redundancy Check (CRC) is a widely used cyclic code.

Polynomially generated codes are further divided into the Bose-Chaudhuri-Hocquenghem (BCH) codes, and the Golay codes. BCH codes are a generalization of the Hamming codes that enable multiple error correction capabilities and can use an arbitrary alphabet, not just binary. The non-binary class of BCH codes are the Reed-Solomon codes. Fig. 2.2 depicts a clear taxonomy of the FEC codes.

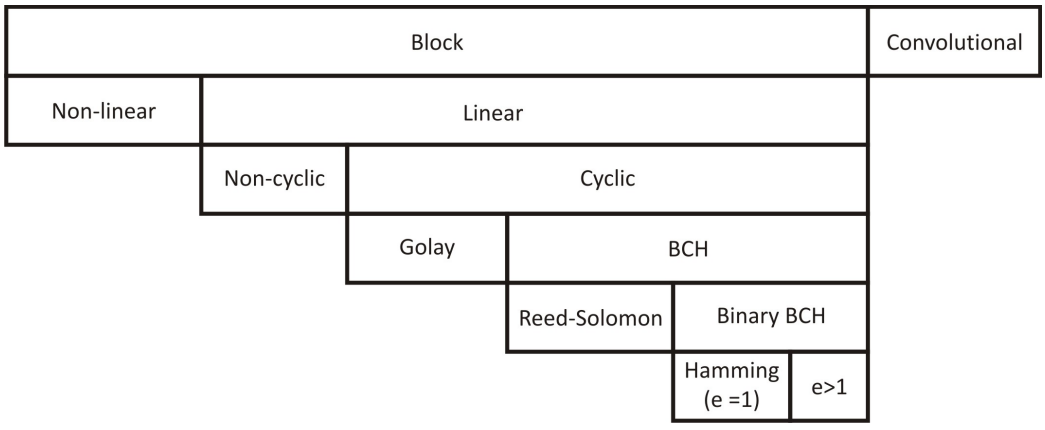


Figure 2.2: Taxonomy of FEC codes. (Adapted from: [37])

Convolutional codes are a different class of codes that besides the current input block of size k use the preceding m blocks (of the same size) to form the n-bit code.

Thus, they are denoted as (n, k, m) codes. m is also called the constraint length. Encoder design is extremely simple, using several registers and modulo-2 adders (e.g., XOR gates). Nevertheless, decoding is more complicated. There are three main types of decoding: sequential, threshold (majority logic) and Viterbi decoding. By far most popular and used is the Viterbi decoding [119].

Checkpointing, rollback and retry

Error codes could sometimes only detect errors and do not correct (all of) them. There are also mechanisms that are designed only to detect errors (see Example 3 and the following Subsections). The question is what to do once the error is detected?

In digital telecommunications, ARQ (Automatic Repeat Request) mechanisms are used to inform the sender that the received message is erroneous, and request resending the same message. Basically, there are three mechanisms: stop and wait, go-back-N and selective repeat. In the *stop and wait* protocol each block of data is acknowledged by the receiver. Transmission of the next block proceeds only if a positive acknowledgement for the previous block is received. If the block is timed out (no acknowledgement is received) it is automatically retransmitted. In the *go-back-N* and *selective repeat* protocols each block has a sequence number. Each acknowledgement contains the sequence number acknowledging all blocks up to $N - 1$. If a negative acknowledgement is received, or a block timed out, all data blocks starting from the lastly acknowledged block are retransmitted. In the selective repeat version only negatively acknowledged (or timed out) blocks are retransmitted. Thus, selective repeat is more flexible and offers higher efficiency, but does not have well defined storage (memory) requirements, which can be large. On the other side, go-back-N has well defined storage requirements.

Nevertheless, in control and processor-based systems the ARQ mechanisms are not appropriate. A widely used technique here is *Checkpointing, Rollback* and *Retry* (CRR): periodically save the system state and retry operation when error is detected by rolling back to a previously saved error-free state. CRR was shortly introduced in Section 1.1. Nevertheless, since this technique is important for the work in the thesis, a more elaborate explanation is given here.

In a digital system, the **system state** is defined by the contents of the system's storage elements (e.g., flip-flops and memory cells). Most often, the input values of the system also have to be considered as a part of the system state. Thus, it is clear that in large systems the system state can be huge – hundreds of megabytes. Nevertheless, not every bit of information is usually necessary to properly restart operation, i.e., a reduced system state could be specified. Therefore, a **checkpoint** is defined as a snapshot of the information needed to restart system operation at the moment it was taken. Checkpoints have to be saved on a medium with sufficient reliability (e.g., RAM memory with fault-tolerant mechanisms and battery back-up).

Checkpoint latency is the time needed to save the checkpoint. *Checkpoint overhead* is the extra time added to the operation (execution) time. Overhead and latency are identical in simple systems. Nevertheless, system operation could be (partially)

overlapped with checkpointing, thus making the overhead smaller, or practically close to zero.

Introducing a CRR technique requires many issues to be resolved. E.g., how frequent and at which points checkpoints should be taken? How to reduce overhead? How to checkpoint if the system is distributed (in order to avoid deadlocks or repeated rollbacks)? How transparent, i.e., at what level (kernel/user/application) should checkpointing be? What to do if multiple retries fail – report permanent faults and halt operation?

Error control coding in (multi)processors

Desktop, server and embedded systems have different requirements regarding dependability and fault tolerance. For example, embedded systems used in automotive, medicine, aircraft or spacecraft applications are safety- and/or mission-critical and require high fault tolerance; embedded systems used in satellites have to sustain long-life operation. Server processors, on the other side, must be highly-available, since their downtime is very expensive. The average downtime costs vary considerably across industries, from approximately $90.000 per hour in the media sector to about $6,48 million per hour for large online brokerages [31]. Therefore, a special attention is given to the Reliability, Availability and Serviceability (RAS) features of server processors. At last, desktop processors are low on fault-tolerant mechanisms because of the lower dependability demand. Usually only parity and simple ECC are used for caches and memory controllers.

The RAS features of the server multiprocessors **IBM Power6** and **Power7** include numerous error detection, correction and recovery techniques, control and data-flow checkers that cover almost all SRAMs and registers [93, 43, 11]. All I/O signals are protected by ECC. Internal signals are also ECC protected. Almost all data and control signals are equipped with error detection. The control units employ logical consistency checkers that check whether the states are valid with respect to their state machine. When a core error occurs, the instruction is retried. A low-level CRR is done by a recovery unit that marks whether the instruction completed without errors. In a super-scalar architecture like POWER, more than one instruction can complete at the same time. Thus, checkpointing has to be done after successful completion of a group of instructions. The core restarts execution due to a core fault from the last checkpoint. Nevertheless, this is only effective for transient errors. If a permanent error is detected, recovery is done at higher level. The task is assigned to another core, isolating the faulty one. Fig. 2.3 shows the fault-tolerant features of the Power6 multiprocessor.

The L1 caches are parity protected, while L2 and L3 caches are ECC protected. This is because ECC is slow in comparison to the L1 memory and takes large area i.e., it is not cost-effective to implement ECC for memories with small latency. In case of a parity error at the L1 cache, the block is fetched again from memory. The L1 cache (consisting of a 4-way and 8-way set-associative instruction and data cache, respectively) implements a mechanism for permanent fault recovery. If a permanent fault is detected in a set, the set is marked as unusable. L2 and L3 caches include

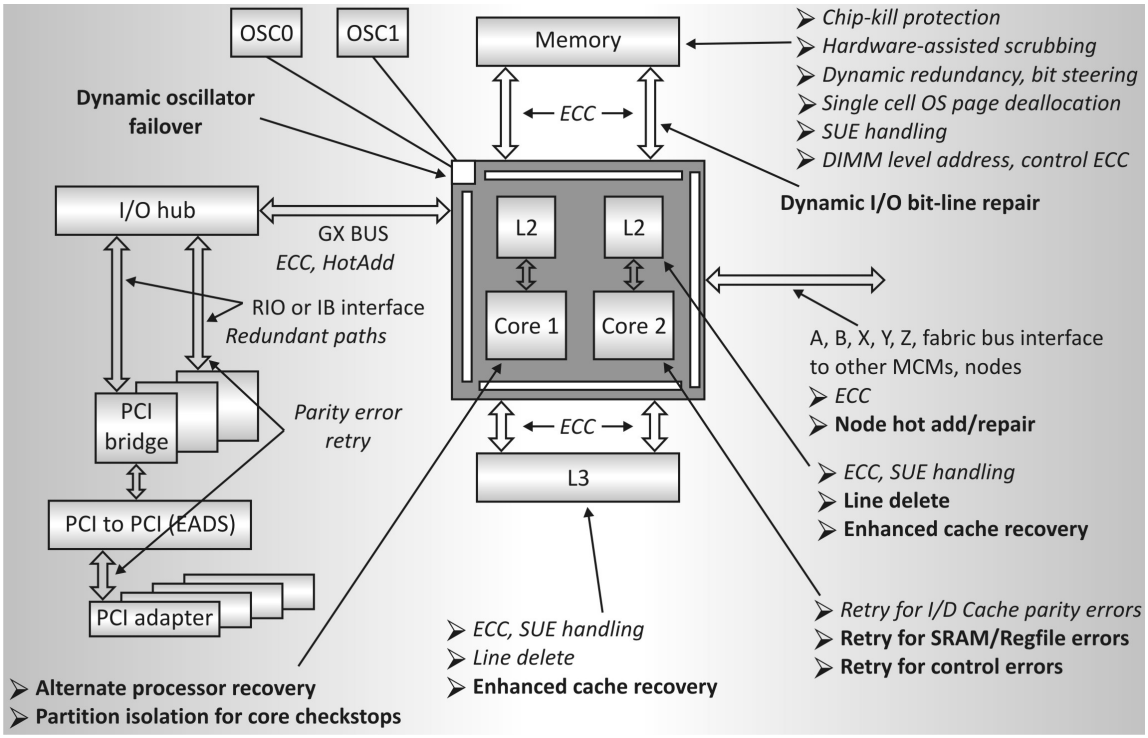


Figure 2.3: RAS features of the IBM Power6 multiprocessor. (Source: [93]) The features in bold are enhancements over the Power5 processor. The Power7 processor builds upon Power6 by adding features like redundant service processor with dynamic failover and active memory mirroring.

cache line delete and enhanced error recovery for multibit errors. All caches have special tags and flags for faulty data which handle non-correctable errors. I/O lines can be dynamically repaired. Spare lines and pins are added, so if a wire or pin is faulty, the error is reported as correctable and the data flow is redirected through the spare pin. Power 6 has also a redundant oscillator which runs concurrently with the active one. They are kept in phase, so if the active oscillator fails, the other takes over without affecting operation. In a case of a permanent fault, the processor isolates itself and the workload is moved to another processor. POWER7 further extends the error resilience by using a stacked latch design, which is inherently less prone to single event effects [43].

A plethora of parity, SEC-DED and other types of ECC are used in the **SPARC64 V** [4] and **Sun Niagara II** [105] server multiprocessors. On the other hand, the **AMD Opteron** has only a parity protected instruction cache, ECC protected data and L2 cache, and ECC support for the DRAM memory controllers [14].

The **ARM Cortex** is a highly customizable embedded processor for real-time systems with fault-tolerant options [10]. The data and instruction caches have parity and ECC build options which may be included during implementation. Furthermore, a second, redundant copy of the core that shares the input pins with the master core

may be also implemented. The master core always drives the outputs. Nevertheless, a comparison logic may show disagreements between the master and the redundant core, thus indicating errors.

Leon3-FT [35] is a fault-tolerant version of the LEON3 processor used in embedded applications. It has an instruction and data cache of 8KB. Both tag and data parts are protected with four parity bits per 32-bit word, detecting up to 4 errors per word. If an error is detected, the corresponding cache line is flushed and the instruction is restarted. This introduces a penalty of additional 6 clock cycles. There are also error counters for detected and corrected errors which can be used for analysis. The register file has four RAM blocks protected by four parity bits per 32-bit word, plus a duplicated copy of the original data word. Upon a detected parity error, the copy of the data is read out from a redundant location in the register file, replacing the failed data. This operation takes place during normal pipeline operation i.e., without restarting the pipeline. Data correction is transparent to the software and does not incur any timing penalty. If the redundant data also contain errors, a register-file error-trap is generated. Error counters that monitor register file errors are also provided. **Leon4-FT** further provides Reed-Solomon protection of the DDR2 interface.

RH32S is a radiation hardened 32-bit processor for embedded applications [54]. Fault-tolerant features include microinstruction retry on error detection, cache line refill and software rollback. All functional units are monitored for errors. A recovery procedure is invoked on a detected error. All of the internal pipeline stages and control registers are byte parity protected. The memory interface has a SEC-DED facility. L1 and L2 caches use parity error detection and line invalidation. All cache words including tags have four parity bits. If a parity fault is detected during a cache read, it is treated as a miss, and a cache line refill is performed.

In [118], the authors propose a technique for online error detection and correction of erratic bits in register files.

Error control coding in on-chip memories

In the previous overview of error control coding in processors, it was obvious that each of the reviewed examples (even desktop processors) have mechanisms for protecting the caches and the memory controllers. This is not by a coincidence, actually the memories (both on- and off-chip) are responsible for the largest part of errors in the systems [70]. Furthermore, as discussed in Section 1.3, the percentage of memory area in chips is constantly increasing (see Fig. 1.11). Therefore, memory fault tolerance is of primary importance, and thus, a widely researched topic.

Besides improvements in the production process, materials and layout, the following error mitigation techniques are used in memories at the architectural level. They are all based on error control coding techniques.

Parity and ECC codes combined with memory **cell interleaving** (physically distant placement of logically neighboring memory cells) is a frequently used tech-

nique. Physically distant memory cells of a logical memory location[2] lowers the probability for multiple-bit upsets in that location, thus enabling simpler ECC to be used.

Re-fetch of a clean block is used in some of the previously reviewed processors. Caches which are only parity protected can always fetch a clean copy of the block from a memory at a higher hierarchical level when error is detected.

Memory **scrubbing** is a mechanism in which memory locations are read, the possible errors are corrected by an ECC for example, and than the correct data is written back to the same location.

In a memory **mirroring** technique, a redundant copy of the entire memory content is created, which is accessed through a second channel. If an error is detected in one of the memory copies, the controller continues operation using the other copy without any disruption, and tries to fix and resynchronise the errant copy. An obvious drawback of this scheme is that the amount of memory has to be doubled.

An in-depth explanation of numerous nanoscale memory protection and repair techniques is given in [49]. A vivid example of a multiprocessor with an abundance of memory protection techniques is the **Itanium 9300** server multiprocessor (shown in Fig. 2.4). As discussed, the caches in (multi)processors are also mainly ECC or parity protected. More elaborate schemes for cache protection are also reported. For example, in [63], entire cache block is sacrificed in order to detect and repair errors in other blocks.

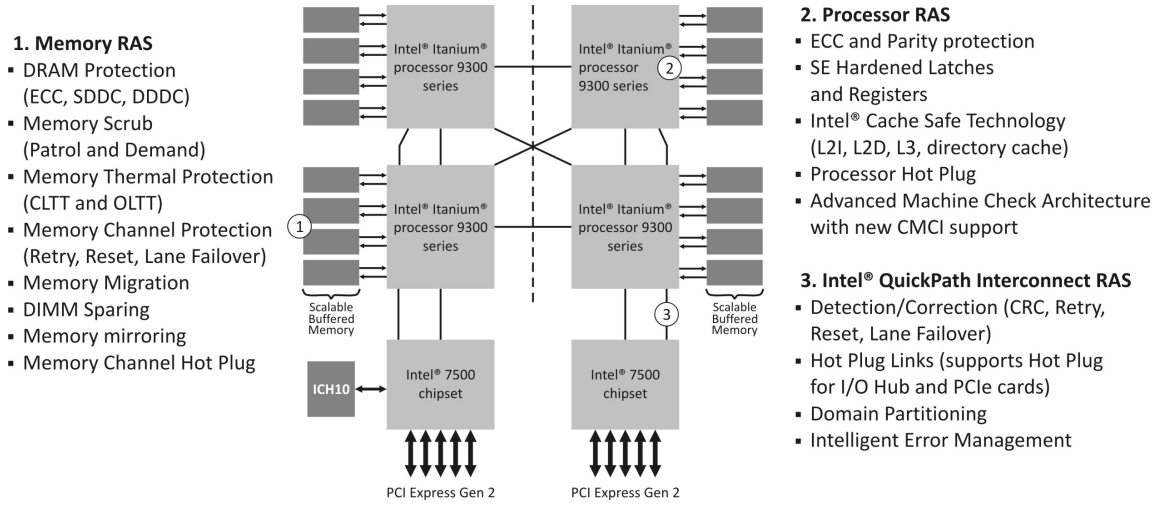


Figure 2.4: RAS features of the Itanium 9300 multiprocessor. (Source: Intel)

Discussion

The previous overview of fault-tolerant mechanisms in processors and memories shows that information redundancy techniques are massively used. This is because they have

[2]A memory location is a logical sequence of bits (memory cells) selected by a single memory address.

relatively lower power and area overhead than the space-redundant schemes discussed in Subsection 2.1.3, and still cope with both transient and permanent faults. They also have lower performance overhead compared to the time-redundant schemes in Subsection 2.1.2. On the downside, information redundancy techniques have lower fault coverage, and much more complex design and verification than techniques based on modular (space) redundancy.

2.1.2 Time redundancy

Time redundancy techniques increase the fault tolerance by performing the same operations twice. If the results in both cases are not the same, a fault is detected, triggering a recovery mechanism (e.g., CRR).

Many of these schemes can only detect transient faults. For example, a permanent fault in a circuit for addition will always produce the same erroneous result no matter how many times the operation is performed. Another disadvantage is that a significant performance overhead may be introduced by performing the same operations twice. Power consumption is also increased. Furthermore, these techniques only detect errors and start recovery, i.e., uninterrupted operation in presence of errors is not possible like in ECC techniques. On the positive side, an advantage of these schemes is the low, or zero hardware (area) overhead.

The double execution of the same operation could be consecutive or partially overlapped. For partial overlapping, two (not necessarily identical) modules are needed. Moreover, consecutive re-execution could also use two modules. A technique which fully overlaps the operation of two identical modules is actually the Dual-Modular Redundant (DMR) scheme discussed in Subsection 2.1.3. This Subsection makes an overview of fault-tolerant mechanisms that introduce time redundancy.

Multiple sampling

Multiple sampling is a simple circuit-level technique that detects transient faults whose duration is less than Δt (see Fig. 2.5). Two latches/flip-flops are used to sample the signal in different instances of time at Δt distance. A modulo-2 addition of the main and shadow latches/flip-flops shows whether there is a transient fault. Besides the performance overhead which is a function of Δt, this mechanism also introduces additional hardware – at least a shadow latch and a XOR gate per protected signal.

This technique is used in several works, including processor pipelines and high-speed circuits [30, 21]. In [76], the signal is sampled in three different points of time and the final output is selected by a majority voter.

Repeated execution

Repeated execution is a very simple technique in which the same operation is executed by the same module twice, consecutively. Most frequently, this scheme is implemented entirely on the software level without additional hardware, although hardware implementations with minimal area overhead could be also easily realized.

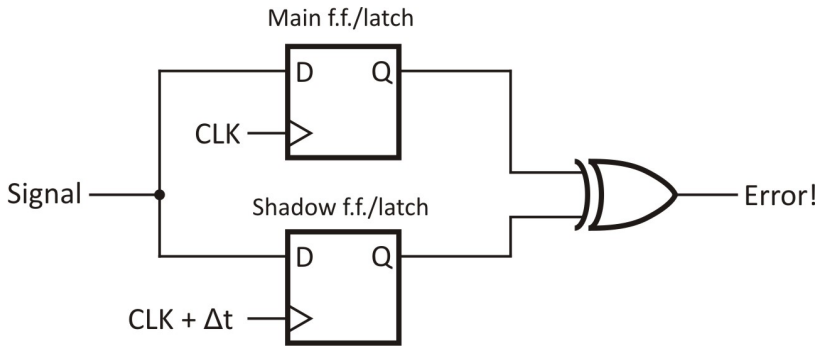


Figure 2.5: A multiple sampling circuit

Performance and power consumption overhead are always present due to the extra execution, comparison and recovery.

Diversely-repeated execution is a simple extension of this technique in which the repeated operation is done differently than the first. E.g., let the first multiplication produce a result $R_1 = A \times B$, and then swap operands, producing $R_2 = B \times A$. The results R_1 and R_2 are compared at the end, potentially showing faults. With this scheme, even some permanent faults could be detected. Furthermore, if there are two multipliers in the system, one may opt to do the multiplication using both of them, which is actually done in [121] with very low area overhead.

An approach based on diversely-repeated execution is presented in [88], where run-time error detectors are derived by examining the application properties extracted using the compiler's facility for static analysis. The derived detectors diversely recompute the values of the identified critical variables in the program.

It is worth mentioning that repeating execution more than twice introduces too much overhead and is rarely used in practice, i.e., it is simply not appropriate even for applications with modest performance requirements.

Redundant execution

The built-in capabilities in (multi)processors for increasing performance could be used to actually increase the fault tolerance. For example, a processing element (PE) that supports multiple threads of execution could be used for redundant execution of the same software. That is, two copies of the program are executed concurrently by the thread facilities of the PE, and the results are compared at the end by software. This partial overlapping of executing the threads can significantly reduce the performance penalty compared to a consecutive (non-overlapped) execution. Nevertheless, looking more broadly and taking into account other programs in the system, the performance capabilities of the PE are in any case reduced, since the redundant thread takes over the place of another independent thread. Thus, there is still a performance penalty. Such a scheme is detailed in [94], were the Simultaneous Multi-Threading (SMT) capability of superscalar multiprocessors is used as a mechanism for transient fault detection.

In a multiprocessor (e.g., a multi-core processor), an entire core could be assigned to concurrently execute a second copy of the program. The discussion is the same as in the multi-threading case. A dual and triple core-level redundant execution where two and three copies of the same program are executed by different cores is presented in [38]. Solely stores are compared in order to reduce performance overhead, which is enough since only store instructions are output instructions in a RISC core. On a mismatch, a CRR technique is employed. The authors report 99% transient fault recovery with a 5,2% performance overhead. Although not the case in [38], an advantage of multi-threading and multi-core based redundant execution is that the fault-tolerant system could be implemented entirely at the software level, using Commercial Off-The-Shelf (COTS) (multi)processors.

A technique based on redundant execution is presented in [110, 109], where programs have the option to be executed in a fault-tolerant manner. A leading core executes instruction chunks which are afterwards redundantly executed by a trailing core. The leading core heuristically marks some of the instructions as critical, and forwards the results of their execution to the trailing core for comparison. Furthermore, the leading core supplies the trailing core with all outcomes of branches and load values, which may speed up execution since the cache is not accessed and the directions of the branches are already resolved. The distinction between leading and trailing cores is only logical i.e., both have identical structure. Nevertheless, this scheme introduces additional pipeline structures in the cores in order to enable the described operation. Interestingly, the authors do not show evaluation results of this approach regarding improvements in fault tolerance, although this is the primary goal of the papers.

A slightly different approach in which the leading and the trailing core additionally make use of another capability of the superscalar processors – the speculation mechanism, is presented in [127]. A detected transient fault can be simply treated as a misspeculation and corrected by the already existing facility for dealing with misspeculations.

A technique for transient fault mitigation in embedded microprocessors is presented in [125]. The processor executes a primary and redundant thread of the same program, and compares the results at the end of the pipeline (see Fig. 2.6). On a mismatch, an exception is triggered starting a recovery procedure. Additionally, the caches and the register file are protected by parity and ECC. The scheme introduces minimal area overhead (a redundant program counter and a comparator). Nevertheless, the instruction fetch bandwidth is halved. The authors claim a 71,5% detection of injected transient faults and 22,9% "naturally" masked faults by the internal logic.

The on-line software testing and monitoring techniques in embedded multiprocessors [45] and manycore processors[3] [32], periodically check for errors in the cores. The test procedure is scheduled for execution by an idle core in order to minimize performance penalties. Nevertheless, these schemes could discover only permanent

[3]**Manycore processors** have large number of cores in a single chip i.e., over 8 or 16, up to several hundred, opposed to multi-core processors which have up to 8 or 16 cores.

faults. Transient faults in the functional application that is running in parallel on another core are not detected.

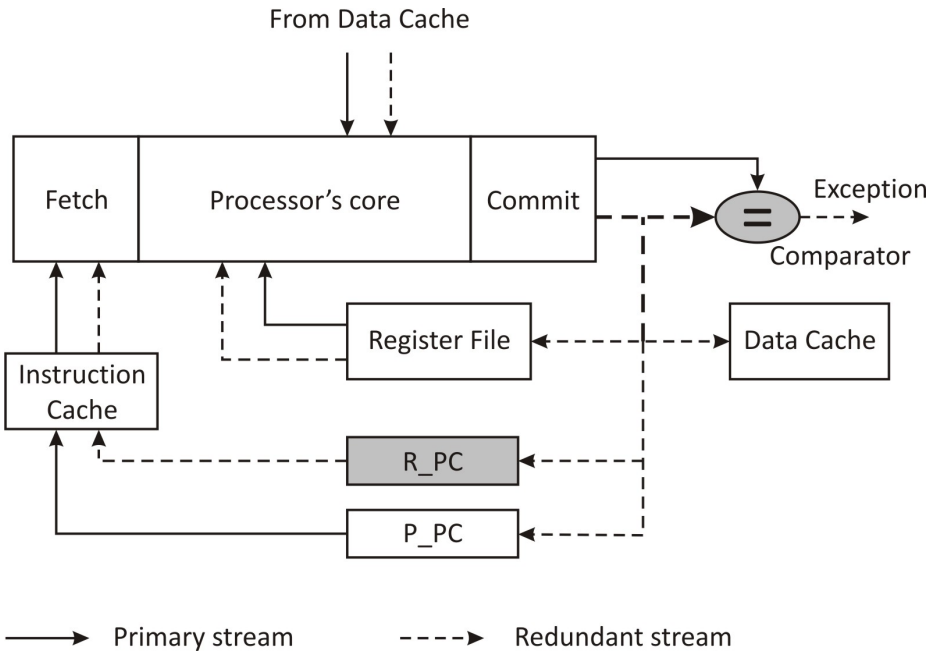


Figure 2.6: Primary and redundant instruction execution. The shaded components are the hardware overhead. (Source: [125])

2.1.3 Space redundancy

Space or hardware redundancy is mainly based on **N-Modular Redundant (NMR)** systems, as the one depicted in Fig. 2.7. Static space-redundant schemes mask faults rather than detect-and-recover, providing uninterrupted operation. This is very important for real-time or timing-critical systems, where time for reconfiguration and recovery cannot be afforded.

Dynamic schemes, on the other hand, switch to spare components upon a detection of a fault, i.e., the system is reconfigured to use a spare (fault-free) module. The hybrid schemes combine static and dynamic operation: they mask faults but also reconfigure the system to use spares.

The downside of this group of techniques is the significant area and power overhead. For illustration, an NMR system has at least N times greater power and area overheads than a single (simplex) module.

Static (passive) redundancy

Fig. 2.7 shows an NMR system with N identical modules $M_0, M_1, \ldots, M_{N-1}$ fed with the same input z. The outputs are expected to be equal i.e., $x_0 = x_1 = \ldots = x_{N-1}$. Nevertheless, in a real case the modules are subject to faults, leading to differences

in outputs. A decision maker D is therefore needed to select the actual output of the system y even in the presence of errors.

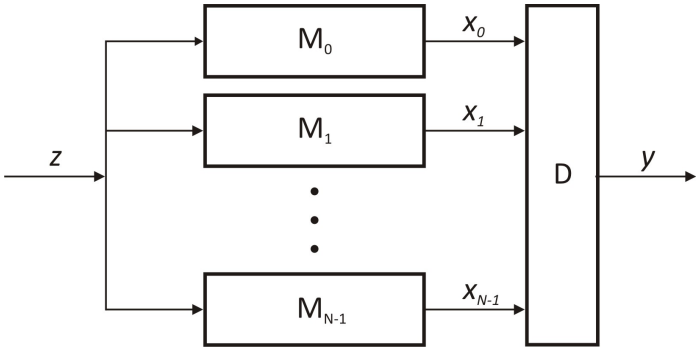


Figure 2.7: An NMR system

A widely used decision maker is the M-out-of-N voter, where of N voter inputs (module outputs), at least M should be equal in order to consider an error-free, uninterrupted operation. Nevertheless, if $M \leq N/2$, ambiguous situations may occur since more than one input value could be a legitimate voting output value. E.g., let in a 2-out-of-4 voter $x_0 = x_1 \neq x_2 = x_3$. Both values of x_0 (equal to x_1) and x_2 (equal to x_3) could be the output of voting. Therefore, one of the most frequently used voters is the **majority voter**, where at least $\lfloor N/2 + 1 \rfloor$ voter inputs (i.e., module outputs) have to be equal for error-free operation, otherwise the system fails.

Evaluating the reliability $R(t)$ of NMR systems is relatively simple. Using the binomial theorem in probability theory

$$P(M \text{ successes in } N \text{ trials}) = \binom{N}{M} p^M (1-p)^{N-M}, \tag{2.1}$$

where p is the probability of success, the following equation is obtained for a Triple Modular Redundant (TMR) system with a majority voter (operation is considered correct if at least two modules are operating):

$$R_{tmr} = \underbrace{\binom{3}{3} p^3 (1-p)^0}_{\text{all modules operating}} + \underbrace{\binom{3}{2} p^2 (1-p)^1}_{\text{2-out-of-3 operating}} = 3p^2 - 2p^3. \tag{2.2}$$

Let each of the modules have equal reliability $R_m(t)$ with exponentially distributed time-to-failure (see Eq. 1.2), i.e., constant failure rate. That is, $p = R_m(t) = e^{-\lambda_m t}$. Furthermore, in Eg. 2.2 the reliability of the voter is assumed to be ideal, i.e., $R_v(t) = 1$ which is not realistic. Let $R_v(t) = e^{-\lambda_v t}$. Thus,

$$R_{tmr}(t) = (3e^{-2\lambda_m t} - 2e^{-3\lambda_m t}) R_v(t) = 3e^{-(2\lambda_m + \lambda_v)t} - 2e^{-(3\lambda_m + \lambda_v)t}. \tag{2.3}$$

According to Eq. 1.5, $MTTF_{tmr} = \frac{3}{2\lambda_m + \lambda_v} - \frac{2}{3\lambda_m + \lambda_v}$. Even if a perfect voter is assumed ($\lambda_v = 0$), $MTTF_{tmr} = \frac{5}{6\lambda_m}$, which is actually lower than the $MTTF$ of a single module $\frac{1}{\lambda_m}$. This pitfall will be explained soon.

Let's first determine the reliability of a general NMR system with M-out-of-N voter, by generalizing Eq. 2.2. That is

$$R_{nmr} = \sum_{i=M}^{N} \binom{N}{i} p^i (1-p)^{N-i}. \tag{2.4}$$

Assuming exponential distribution of the TTF and including voter's reliability

$$R_{nmr}(t) = R_v(t) \sum_{i=M}^{N} \binom{N}{i} R_m^i(t)(1-R_m(t))^{N-i} = e^{-\lambda_v t} \sum_{i=M}^{N} \binom{N}{i} e^{-i\lambda_m t}(1-e^{-\lambda_m t})^{N-i}. \tag{2.5}$$

Table 2.1 shows Eq. 2.5 in unfolded forms for NMR systems with perfect majority voter i.e., $R_v(t) = 1$ and $M = \lfloor N/2 + 1 \rfloor$.

Table 2.1: Reliability and MTTF of NMR systems with perfect majority voters and constant failure rates

N	$R_{nmr}(t)$	$MTTF_{nmr}$
1	$e^{-\lambda_m t}$	$\frac{1}{\lambda_m}$
2	$e^{-2\lambda_m t}$	$\frac{1}{2\lambda_m}$
3	$3e^{-2\lambda_m t} - 2e^{-3\lambda_m t}$	$\frac{5}{6\lambda_m}$
4	$4e^{-3\lambda_m t} - 3e^{-4\lambda_m t}$	$\frac{7}{12\lambda_m}$
5	$6e^{-3\lambda_m t} - 7e^{-4\lambda_m t} + 2e^{-5\lambda_m t}$	$\frac{13}{20\lambda_m}$
6	$15e^{-4\lambda_m t} - 24e^{-5\lambda_m t} + 10e^{-6\lambda_m t}$	$\frac{37}{60\lambda_m}$

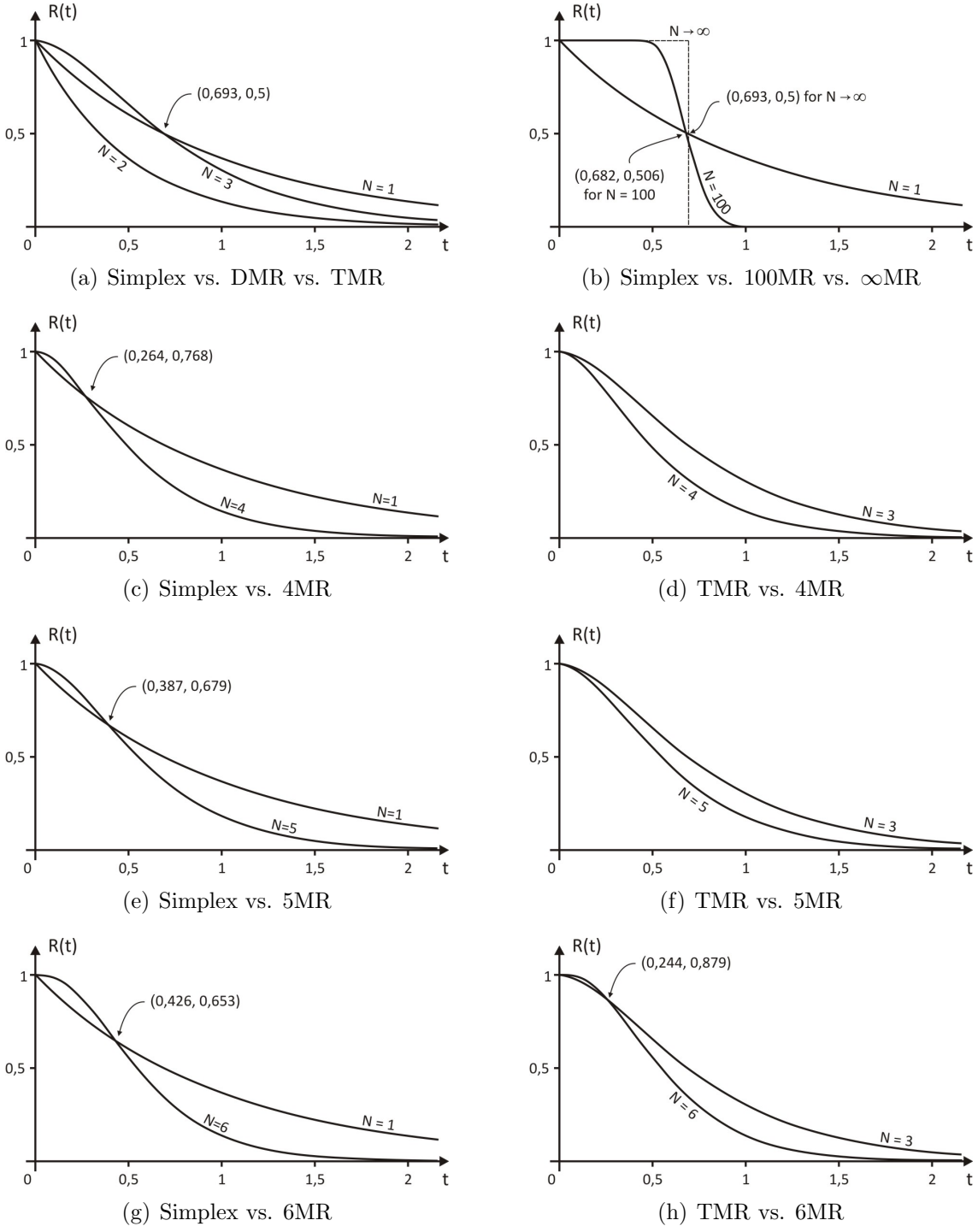
Fig. 2.8 plots the $R_{nmr}(t)$ equations from Table 2.1 for a module failure rate of $\lambda_m = 1$, which is 1 failure per 10^9 hours of operation ($MTTF = 10^9$ hours).

Fig. 2.8(a) shows that a DMR system always has lower reliability than simplex. This is not surprising, i.e., it is expected that a 2-out-of-2 system with two identical modules is more likely to fail than a system with one such module.[4] Nevertheless, the system with two modules can detect errors and trigger recovery mechanisms, while the simplex system cannot. Thus, the equation for DMR systems in Table 2.1 is incomplete.

Fig. 2.8(a) further shows that a TMR system has greater reliability than simplex up to a specific point of time. The two formulas for TMR and simplex are equal at the intersection point. Solving the equation $e^{-\lambda_m t} = 3e^{-2\lambda_m t} - 2e^{-3\lambda_m t}$ gives $t = ln2/\lambda_m$. Replacing this into $R(t) = e^{-\lambda_m t}$ gives $R(ln2/\lambda_m) = 0{,}5$. In the same way, a 4MR system has greater reliability up to $t = ln(6/(1+\sqrt{13}))/\lambda_m$, i.e., intersection point (0,264, 0,768), etc.

Thus, a TMR system for example, has greater reliability than simplex for mission times of up to $ln2 \times MTTF_{simplex} \approx 0{,}693/\lambda_m$, i.e., up to $\approx 70\%$ of the simplex mean time to failure. Looking from another angle, TMR has lower reliability than

[4] In 1-out-of-2 system, $R(t) = 2e^{-\lambda_m t} - e^{-2\lambda_m t}$. $MTTF = 3/(2\lambda_m)$ – 50% better than simplex.



(a) Simplex vs. DMR vs. TMR

(b) Simplex vs. 100MR vs. ∞MR

(c) Simplex vs. 4MR

(d) TMR vs. 4MR

(e) Simplex vs. 5MR

(f) TMR vs. 5MR

(g) Simplex vs. 6MR

(h) TMR vs. 6MR

Figure 2.8: Comparison of reliability functions of NMR systems for various N, with a simplex and a TMR system. ($\lambda_m = 1$).

simplex if the single module reliability $R_m(t) < 0{,}5$. Of course, the discussion applies if exponential distribution of the time-to-failure (constant failure rate) is assumed.

It is interesting to see what happens when $N \to \infty$. Fig. 2.8(b) shows a plot for $N = 100$. According to 1.5 and replacing Eg. 2.5,

$$MTTF_{N\to\infty} = \lim_{N\to\infty} \int_0^\infty R_{nmr}(t)dt = \frac{ln2}{\lambda_m}.$$

Theoretically, an infinite modular redundant system is perfectly reliable up to $t = ln2/\lambda_m$, or, $R_m(t) > 0{,}5$. Afterwards, it is useless. (Again, this applies for exponential distribution).

Plots 2.8(d) and 2.8(f) show another pitfall: 4MR and 5MR always have lower reliability than TMR. 6MR has only slightly greater reliability than TMR (up to 24% of simplex MTTF). In other words, greater N does not necessarily mean more reliable system.

The discussion that comes from these calculations and graphs can be a little bit misleading. An intuitive example is the following. Two faults in two different modules in a 5MR system will not fail the system, but will surely fail a TMR system, which is not caught by Fig. 2.8(f). This comes from the basic definitions of reliability and failure rate (see Subsection 1.1.4) that actually treat only permanent faults: if a module fails at time t, it is considered useless after that time. Nevertheless, modules in NMR systems can temporarily fail due to a transient, but afterwards continue operation. Taking into account transient faults will significantly improve the reliability picture of NMR systems. Of course, the reliability of recovery and repair mechanisms should be also considered in order to completely characterize the overall system reliability.

So far, the analyses mainly assumed perfect voters which is never the case in a real system. If the voter in an NMR system fails, the entire system fails. Fortunately, voters have relatively small complexity and area, which lowers the probability of voter faults. Nonetheless, there are also mechanisms for voter protection. A totally self-checking NMR system with concurrent error location capability is presented in [57]. The system determines whether an error occurred during voting as well as its location. A full error coverage in the system is achieved, i.e., the error could be detected in the redundant modules, the voter, or the error detection circuit. The authors compare their work to a similar scheme proposed in [36]. Another technique based on error correction by Alternate-Data Retry for increasing NMR voter reliability is introduced in [113].

Solutions based on NMR are widely used in processors, starting from DMR/TMR protected pipelines [78, 114] and TMR protected registers in the Leon3-ft [35, 16], to entire TMR processor [53].

The lock-step architecture is a DMR technique that uses two processors: a master and a checker which synchronously execute the same instructions. Only the master has access to the memory hierarchy and actually drives all outputs, while the checker monitors the buses and executes the instructions fetched by the master. Both master

and checker outputs are compared. If the values disagree, an error is detected, but the faulty processor can not be identified. A loosely-synchronized lock-step architecture does not have this strong relationship between the processors. The independently-operating processors have access to different memory sub-systems. A subset of critical tasks (or all tasks) are duplicated in both memory spaces and executed in parallel. Outputs are exchanged and compared by both processors. If outputs disagree, they do not commit and an error recovery procedure is invoked. In [15], a combination of these two techniques is presented – dual lock-step architecture. Actually in the loosely-synchronized architecture each of the two processors are replaced by a master and checker processors accordingly. The memory sub-system is divided in four banks interconnected by a crossbar switch. The authors also give comparison between these architectures (plus a TMR architecture) in terms of area and performance. However, reliability analysis are shallow.

A comparison of TMR ALU (Arithmetic/Logic Unit), DMR ALU with recovery, and parity-checked program counter with recovery in a 32-bit MIPS compatible processor is presented in [66], with 84,6%, 84,2% and 78,2% masked faults, and 1,64x, 1,72x and 1,07x gate count overhead, appropriately. (The unprotected system masks 68,8% of the injected faults.)

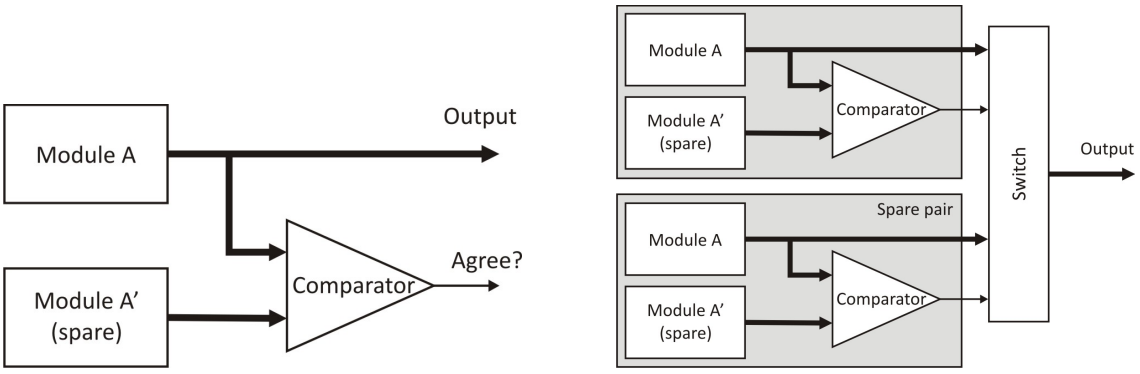
A low-level DMR register protection against SEEs is presented in [102]. Based on this technique, an automated procedure for generation of the redundant modules and checkers is presented in [89]. In [12], only specific input combinations are TMR protected i.e., a selective fault tolerance is applied in order to lower area costs. Protecting 50% of the input combinations for example, gives an area reduction of 20%.

Dynamic (active) redundancy

In dynamic redundancy schemes, spare functional blocks are activated when a fault is detected. Alternatively, the system could decide to dynamically "offline" a redundant functional module upon detection of a fault, and later decide to use it again. Thus, in contrast to static schemes, dynamic schemes reconfigure the system. The automated switch to a redundant module is called *failover*. A similar term is *switchover*, which denotes the case when human intervention is required. At last, *failback* (*switchback*) is the process of restoring a system in a failover (switchover) state, back to its original state.

The spares in the system could be kept unpowered (cold), or powered (hot). Cold spares extend system lifetime since aging effects affect only the powered circuits, as discussed. Besides that, a single cold spare doubles the MTTF of the system given that faults are always detected and the reconfiguration circuit never fails. A drawback of cold spares besides the inactive area (hardware), is the additional power up and initialization time. Furthermore, a cold spare cannot help detecting faults i.e., the active circuit has to employ time or information redundant techniques, or, periodically do offline tests in order to detect faults. Hot spares, on the other hand, can be used to detect faults. Fig. 2.9 shows two examples of schemes using dynamic redundancy. The duplicate-and-compare system in Fig. 2.9(a) is similar to a DMR system, with the difference that upon fault detection a diagnostic procedure is invoked to pinpoint the

failing module, which is afterwards replaced by a hot spare. The pair-and-spare system in Fig. 2.9(b) builds on this, further enabling uninterrupted operation during the diagnostic procedure, which is suitable for timing-critical applications. Nevertheless, opposed to cold spares, hot spares age similarly to other modules in the circuit.



(a) Duplicate-and-compare (b) Pair-and-spare

Figure 2.9: Dynamic redundancy using hot spares

Schemes based on dynamic redundancy are broadly used in multiprocessor systems, especially in multi-core and manycore processors. For example, apart of the other RAS features, the highly-threaded Niagara II server processor includes a dynamic management of faulty cores and thread facilities. The thread facility is dynamically "offlined" if it frequently experiences errors. If more thread facilities in the core further experience errors, the entire core is offlined [105]. A comparable strategy for faulty core isolation in commodity multi-core processors is presented in [2].

Manycore multiprocessors usually have several spare cores on-chip in order to increase yield, sometimes drastically [64]. Moreover, spares can be activated later, if permanent faults are diagnosed [124, 72], i.e., the faulty cores are replaced by the spares. Reconfiguring a manycore processor in this way gives rise to several problems. For example, the NoC (Network-on-Chip) infrastructure is changed when a core is offlined and replaced by a spare. Thus, programmers may have a hard time developing operating procedures that efficiently adapt and map the applications to the dynamically changing network topology. A downside of these manycore schemes is the incapability of masking transient faults, unless additional mechanisms are provided.

Hybrid redundancy

Techniques based on hybrid redundancy combine static and dynamic schemes. TMR with spares, self-purging redundancy and NMR On Demand (NMROD) are the most representative hybrid schemes. In a TMR with spares scheme, a failed TMR module is replaced by a spare (either hot or cold) without interrupting operation. Of course, this could be generalized to NMR with K spares. The self-purging redundancy, or, sift-out scheme presented in Fig. 2.10 uses N redundant modules and a M-out-of-N voter where M is usually 2. All modules that produce an output different than the

voter output are self-purged, i.e., disconnected from the system by the elementary switches. Using the initialization inputs of the elementary switches, the system can bring the disconnected modules back to function.

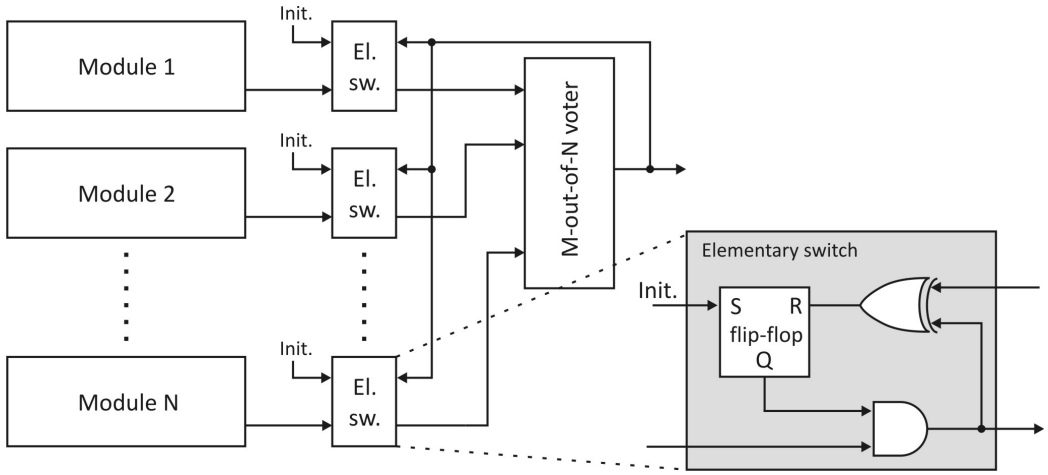


Figure 2.10: Self-purging redundancy

The NMROD scheme may have different implementations. Usually power consumption is taken into account leading to the following pattern NMROD pattern. Normally, a DMR system is formed comparing the outputs of two redundant modules. All other redundant modules are powered off. On a mismatch, the system powers up another module and forms a TMR system, and repeats the failed procedure. Then, the system may failback or power up additional modules. Of course, in an adaptive scheme (like the one presented in this work) the system may decide on the number of redundant modules according to the application requirements, besides the considerations of power saving and reducing aging.

An NMROD technique employed in processors is presented in [91, 92], where a pool of computational units (C-units) and control (voter) units is formed. Issuing an instruction triggers allocating an appropriate voter which further assigns two C-units to redundantly execute the same instruction. The two C-units return results to the voter. If they agree, the instruction is committed and the allocated voter is released. Otherwise, the voter incrementally allocates C-units and compares the results until two of them agree.

One last thing to mention regarding dynamic and hybrid redundancy is that the reliability analyses can be performed similarly to the static case. Including these analyses here would unnecessarily lengthen the discussion.

2.2 Reducing aging and power consumption

A synonym phrase to "reducing system aging" is "increasing system lifetime". Both of them are used interchangeably in the thesis. Occasionally, the term "**wear-out**" could be found as a synonym to aging. On the other side, *power* which is the rate

at which work is done, is different from *energy*, which is the power expended over time. In other words, energy is power integrated over time, or alternatively, power is the instantaneous energy. Although power and energy have clear definitions, they are often misused – consider Example 6. In the thesis, reduction of power consumption is one of the objects of investigation.

Example 6. *Processor 1 with a power consumption P does a given job in a time period T. Processor 2 with a power consumption P/2 does the same job in a time period 2T. Processor 1 has twice the power consumption of processor 2, but in both cases the energy required to do the job is the same. Thus, the Power-Delay Product (PDP) is the same, while the Energy-Delay Product (EDP) is two times greater for Processor 2.*

As already underlined in the motivation Subsection 1.4.1 (based on the material in Subsection 1.2.1 and Section 1.3), reducing aging effects and power consumption could be achieved by doing less work i.e., by lowering performance. When the circuit is powered (under stress), it ages and consumes power.[5] In other words, aging and power consumption are strongly correlated.[6] Therefore, all "**de-stress**" solutions that reduce power consumption reduce aging too, and vice versa. Nonetheless, as will be shown soon, there are solutions that take into account only aging or only power consumption, but also both of them.

2.2.1 Reducing power consumption

Power reduction is an "old topic" for which an abundance of techniques are developed over the years. Three of them are massively used in processors, especially in battery-operated, mobile devices. These are **power and clock gating**, and **DVFS** (Dynamic Voltage and Frequency Scaling). Power/clock gating techniques cut off the power/clock of the temporarily unused modules in the system. In a DVFS scheme, the voltage which is quadratically related to power consumption (see Eq. 1.9), is dynamically lowered to save power, or elevated to increase performance. The voltage column V in Table 1.1 shows that each processor after year 2000 has a voltage range specification for the purposes of DVFS. Voltage scaling has to be accompanied by frequency (performance) scaling, since the transistor input-to-output delay is inversely proportional to the supply voltage.

The IBM Power7 multiprocessor has an elaborated power management system. Clock gating and DVFS are the key techniques used to implement three low power states: nap, sleep and heavy sleep [34]. These states introduce different trade-offs between power and core activation latency. For example, the nap state is optimized for fast wake-up: the frequency of the core is reduced, the execution units are clocked-off, while the caches remain coherent. In the sleep state, entire core is clocked-off including the caches. The sleep state saves more power but core activation requires

[5] This seems to be true for other systems, e.g., the human body as an example of biological system ages faster under stress.

[6] An interesting investigation would be to compare the age of Processor 1 and 2 from Example 6, immediately after each of them finishes the job.

more time due to the required initialization and bringing the caches to a coherent state. That is, core activation latency is increased. In the heavy sleep mode, all cores go to the described sleep mode, at a reduced voltage. IBM states that dynamic power savings go up to 50%.

2.2.2 Reducing aging

Aging-reduction technique based on DVFS and micro-architectural adaptation of the microprocessor is found in [106]. The authors present an architectural model and implementation of microprocessors that dynamically track lifetime reliability, and respond to the changes in workload, application requirements and behaviour. Furthermore, a RAMP (Reliability Aware MicroProcessor) methodology is proposed, which is used to estimate the microprocessor lifetime reliability.

Another de-stress technique based on a dynamic scheme where inactive spare blocks are used both for self-repair and lifetime improvement is presented in [62]. A further extension is given in [61], where the multiple functional blocks in VLIW (Very Long Instruction Word) processors (see Fig. 2.12) are used to dynamically improve reliability and lifetime.

A lot of work is done in the area of lifetime-aware task mapping and scheduling in multiprocessors, especially in manycore NoC-based processors. In [51, 52], lifetime-aware scheduling and task allocation for MPSoC (Multiprocessor System-on-Chip) is proposed. Similar schemes used in multimedia MPSoCs are given in [27, 26]. Virtually all of these solutions rely on theoretical models of aging when devising the mapping/scheduling algorithms. Analytical results are obtained either exercising the models, or, by simulation. These proposals take into account only the aging-related aspect of reliability i.e., SEE fault tolerance is not treated. Some of them do take into account the run-time application dynamism and try to find a solution based on the power-performance trade-offs [18, 24]. An increase of 16% to 30% in lifetime is reported in these works.

Significantly closer approach to this thesis regarding aging-based task scheduling is the one described in [87]. The paper presents an adaptive idleness distribution technique that tries to equalize the lifetime of each core in MPSoC platforms. The authors assume that process variations and variations in aging effects will lead to uneven performance of the cores. Thus, a possible solution is to idle the cores, each with appropriate duty cycle. However, the authors rely on error detection and correction mechanisms as means to monitor aging. That is, age information is obtained by counting errors in each core, which number dynamically determines the scheduling duty cycle. This can be misleading since errors may be introduced by SEEs too, not only by aging effects.

Alternative parameter that could be used to monitor aging and make decisions about task mapping and scheduling is temperature. The work in [26] advocates task mapping relying on temperature distribution. Although temperature is strongly correlated to aging, this parameter alone could be also misleading. Namely, factors like supply voltage, current density, architectural characteristics and application requirements are also very important and directly affect aging. Motivated by these

observations, the work in [42] proposes a task mapping technique based on ant colony optimization. Having the system description and an initial task graph, various solutions (ants) are synthesized. The information about the solutions is shared by pheromones. As ants in the real world use pheromones to direct other ants to food sources, here too, each synthesized solution directs other solutions to the optimal task mapping. Although the authors report an average of 32% lifetime improvement over temperature-driven schemes, this mechanism does not take into account run-time application dynamism, and is relatively tedious for implementation.

On the other side, the work in [33] bases task mapping on an array of circuit-level wear-out sensors. The scheduling and mapping policies are dynamically built according to the feedback from these sensors and the assessment of the current workload. The authors report a 38% lifetime improvement of a 16-core multiprocessor using their technique, over a naive round-robin scheduler.

In the thesis, lifetime-aware task mapping and scheduling is also based on special on-line HCI and NBTI monitors [SKK11b] which are placed in all cores in the multiprocessor. Nevertheless, primary mechanisms of lifetime improvement are clock and power gating, in a special de-stress mode of operation. This mode also uses the feedback from the aging monitors.

2.3 Dynamic adaptation to application requirements

The thesis title hints a multiprocessor technique that dynamically adapts to the application requirements regarding aging (lifetime), fault tolerance to SEEs, performance and power consumption. The motivation behind this work was clearly highlighted in Subsection 1.4.1, based on the considerations exposed in Chapter 1. Such a technique is suitable for long-life mission critical systems which require real-time, or, timing-critical operation. An example is a satellite multiprocessor involved in many tasks concurrently. Of course, fault-tolerant systems of other types may find this work useful (see classification of fault-tolerant systems in Subsection 1.1.1).

This Section gives an overview of such schemes, proposals and techniques, emphasizing the advantages and disadvantages, comparing them to the thesis proposal.

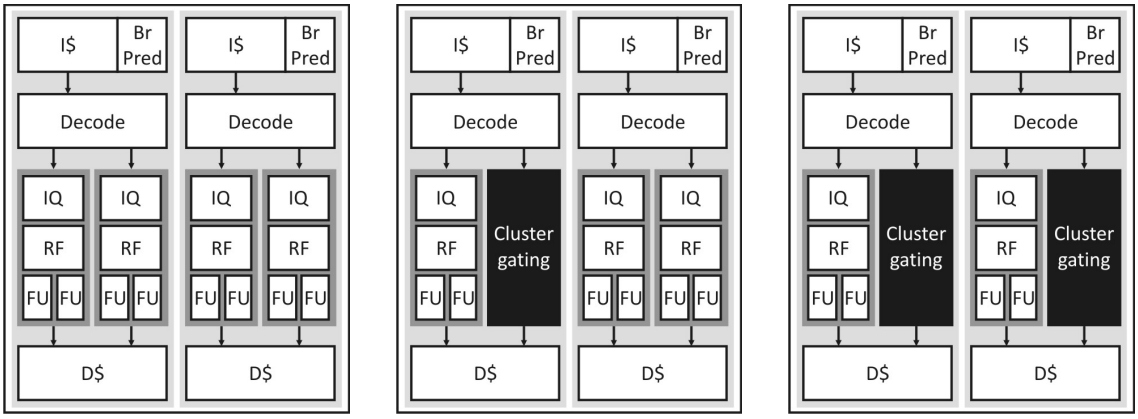
2.3.1 Solutions based on core adaptation

A multi-core processor with a dynamic trade-off between fault tolerance, power consumption and performance is introduced in [98]. It consists of multiple clustered cores shown in Fig. 2.11. When a task with low parallelism requires high-performance, a large core is assigned. On the other hand, for high-performance with high parallelism, several small cores are used. If high performance is not required, a small core is assigned. Making the core small or large is done by powering on/off one or more clusters using special instructions.

Regarding fault tolerance, two modes of operation are proposed, based on the redundant-multithreading technique (see Subsection 2.1.2) i.e., dual large core mode

for high performance and dual small core mode for lower power consumption. Each thread is executed redundantly by the two cores. Similarly, the required performance directs the choice of the operating mode.

The authors evaluate the different modes and processor configurations using the energy, delay, upset-rate product (EDUP). Using rough estimations and architectural level simulation they show for example 21% improvement in EDUP if adapting to application requirements is used, compared to a case when the processor is configured to execute only in the dual small core mode.



(a) Dual large core (b) Hetero core (c) Dual small core

Figure 2.11: Multiple clustered core processor (MCCP). (Source: [98]) Three configurations of two cores with two clusters are shown. Each cluster consists of instruction queue (IQ), register file (RF) and functional units (FU). The cores additionally contain instruction and data cache (I$ and D$), instruction decode and branch prediction unit. Special instructions are used for cluster gating.

In contrast to the MCCP proposal, the fault-tolerant solution in the thesis is based on NMR mechanisms, which is applicable for timing-critical applications where time for error recovery can not be afforded. Also, one of the thesis goals is that the framework could be able to function with virtually any processor core, including off-the-shelf cores (e.g., MIPS, ARM, ARC, LEON), without extending the instruction set. Thus, no special instructions are needed for multiprocessor adaptation. Furthermore, the thesis addresses an additional dependability aspect – multiprocessor lifetime.

Similar in motivation to [98] is the FPSR (Field-programmable self-repair) approach presented in [58]. The authors present an adaptable and self-repairable microarchitecture based on micro-operation units which could be configured by look-up tables (as in FPGA). The programmer directly chooses the micro-operation units and controls the internal operations and instructions. Several instruction sets are supported such as ARM and Texas Instruments' TMS320C55. The objective is that the application extracts the maximum possible fault tolerance with maximum possible performance.

Surprisingly, the authors do not show reliability evaluation of the proposal. In one of the published papers on this subject, only power, area and performance measurements are given. On the other hand, the work in [116] shows a mechanism for average execution time optimization for two fault-tolerant techniques: CRR (Subsection 2.1.1) and majority software voting on several program copies executed by different cores. They report up to 50% improvement compared to the non-optimized case.

The FPSR approach, however, only treats the trade-offs between fault tolerance and high performance. Besides, it requires great efforts in programming and detailed knowledge of the microarchitecture. On the other side, being able to operate with almost any kind of processor core, the thesis proposal leverages both software and hardware development. Already written software (e.g., compilers, operating system) could be reused and extended. The programmer does not need new knowledges and skills.

2.3.2 VLIW-based solutions

In VLIW architectures, as the one depicted in Fig. 2.12, several slots concurrently execute the short instructions packed by the compiler in one long instruction, thus increasing performance. The slots could be alternatively exploited to increase the fault tolerance of the processor [100, 101, 23]. The idea is to perform the same operation by two slots as long as the results match. On a mismatch, a third slot is assigned, forming a TMR system. Actually, this resembles an NMROD scheme described in Subsection 2.1.3. When high-performance is required, the application uses all available slots without replication of the operations, of course, in a non-fault-tolerant fashion.

On the downside, besides the smaller fault-coverage (e.g., error in the control unit could not be covered, which is crucial against SEFI), this approach is applicable only in VLIW processors.

2.3.3 COTS-based solutions

Using COTS cores to build a dependable multiprocessor is an attractive strategy, primarily because high-performance cores which are already proved and verified could be integrated in the planned system, avoiding the effort to design the core itself. Furthermore, a large body of software may already exist for such cores, reducing the effort of software design too. As pointed out in a few places, the thesis proposal is suitable for such a scenario. Of course, there are several other proposals.

A parallel, symmetric architecture for satellite systems built with high-performance COTS components is presented in [126]. To increase dependability, the architecture provides two fault-tolerant modes of operation: centralized and distributed. The system is started in centralized fault-tolerant mode. If errors are frequently detected, the system switches to distributed fault-tolerant mode, where performance overheads are greater, but fault tolerance is higher, thus forming a self-

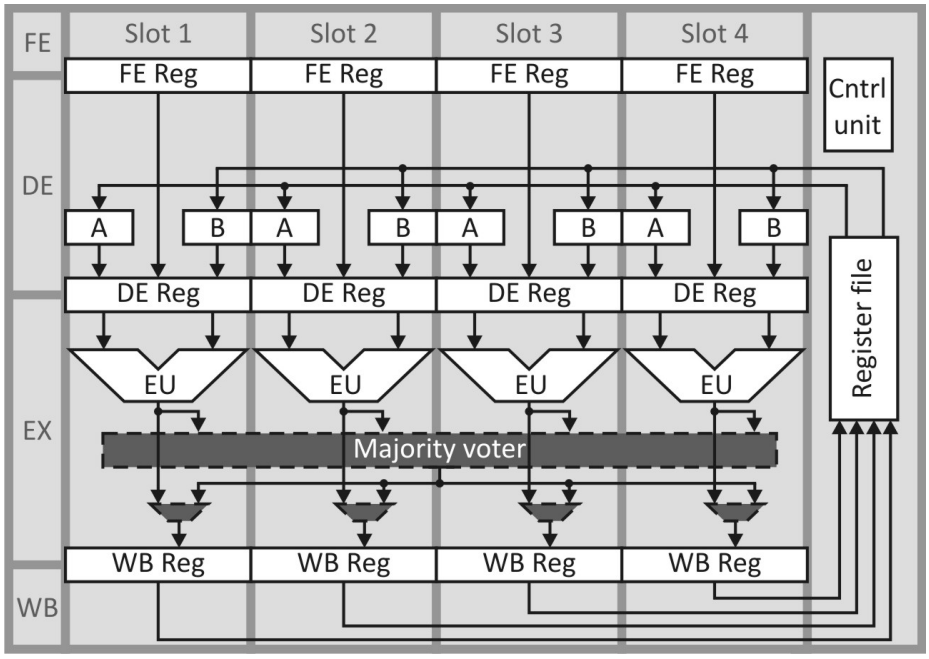


Figure 2.12: VLIW architecture. (Adapted from: [100]) Available slots process in parallel one long instruction which consists of several short (comparable to RISC) instructions. The slots are pipeline replicas starting at the fetch phase, ending at the write-back phase. There is a single register file and centralized control. Note that the proposed reliability enhancement (NMR system, depicted with dashed lines and dark shade of gray) could be implemented at several places in the pipeline – even more rational would be behind the write-back registers. Only data-flow is shown between the blocks, while control flow and instruction supply are omitted for simplicity.

adaptable, hierarchical fault-tolerant scheme. However, here adaptability is related to the number of errors observed in the system, not to application requirements.

Finally, although not adaptable, a dependable multiprocessor based on COTS processors that is worth mentioning is detailed in [48, 97]. As depicted in Fig. 2.13 the cores (data processors 1 to N) are connected with redundant interconnections A and B, and controlled by a redundant, radiation-hardened system controllers A and B. The authors show an irradiation-based evaluation with 90%-95% confidence that the system will be able to fulfill the 120-day space mission, expecting 945 SEUs. Nevertheless, this is a short-mission system that does not employ mechanisms for dynamic adaptation to application requirements and lifetime increase.

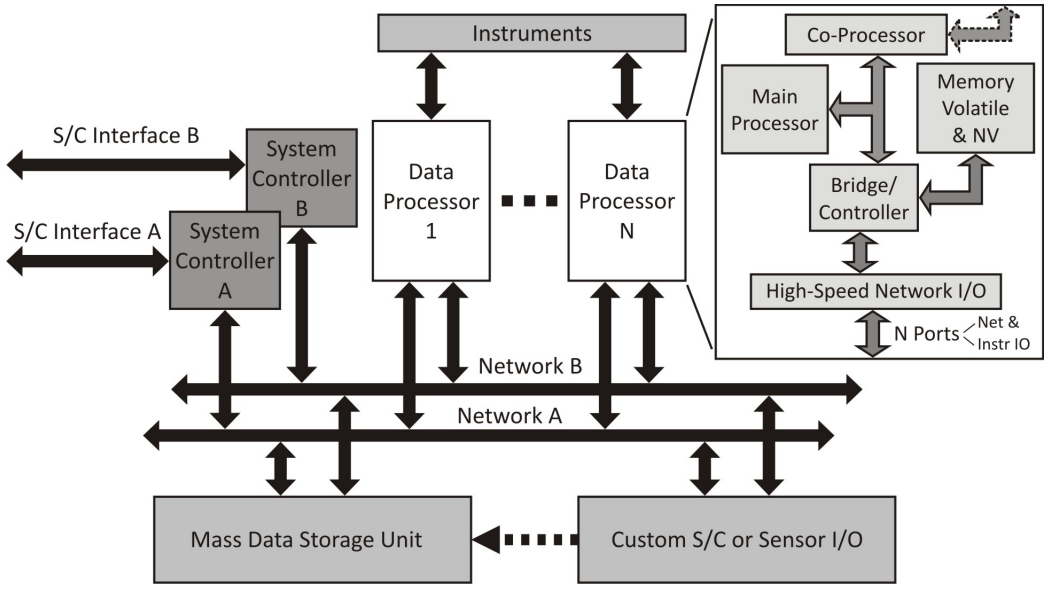


Figure 2.13: COTS based multiprocessor used in space applications. (Source: [48])

2.4 Progress beyond the State-of-the-art

The Sections in Chapter 2, especially Section 2.3 already emphasized many features of the thesis proposal that make progress beyond the current state-of-the-art. Nevertheless, this Section summarizes all the aspects that bring novelty and contributions in the field.

At a first glance, the abstract overview of the thesis proposal shortly introduced in Section 1.4 may seem as a simple combination of known concepts. Looking more deeper, reveals multiple contributions in several fields.

First of all, this work proposes a novel dependable multiprocessor framework targeting fault tolerance, longer lifetime, lower power consumption and high-performance. The key concept is dynamical adaptation of the multiprocessor to application requirements by changing the operating modes (de-stress, fault-tolerant and high-performance). The de-stress mode is based on core gating patterns that transfer the workload to inactive cores, while the currently active cores are deactivated by switching off the power supply or clock; besides de-stressing, this mode enables power saving. The fault-tolerant mode, on the other hand, is based on tightly synchronized core-level NMR, where voting in each clock cycle enables masking faults without interrupting operation, which is important for real-time, or, timing-critical applications. Finally, high-performance mode enables high performance. Chapter 3 describes the proposal in details.

In particular, there are also 10 contributions (see the List of Own Publications at page xvii) regarding the solutions used in the multiprocessor framework, or regarding its evaluation:

- novel aging monitor presented in [SKK11b] and Subsection 4.1.2; a low-complexity, all-digital, self-calibrating monitors are proposed; the cumulative delay effect in an NBTI and an HCI inverter chain is used to make a relative assessment of the age; the ends of the chains are registered, which enables observing the age by interpreting the codes in these registers;

- novel core gating patterns for lifetime improvement based on the aging monitor (see Subsection 3.1.1 and [SKK14a]); after each elapsed gating period, the pattern activates and transfers the workload to the "youngest" core(s) in the multiprocessor, while the currently active cores are powered- or clocked-off; the pattern evaluation method which is based on the Weibul distribution is also novel (see Subsection 6.1.1 and [SKK14a]);

- novel type of programmable NMR voters with self-check capability; a detailed presentation of these voters is given in [SHKK12] and in Subsection 4.1.4; a general method for design is also given in [SHKK12] and in Section 4.4; the "programmable" property enables defining which of the redundant modules should be considered during voting; programming could be done even in each clock cycle; furthermore, the voters output a description of the inputs' state, e.g., which of the modules err (if so); the self-check facility signals whether the voter itself operates correctly;

- novel technique for automated fault injection into the ASIC design flow which is used to evaluate the fault-tolerant mechanisms in the proposal; the technique is presented in [SKK13a] and in Section 5.2; it consists of preparing the gate-level netlist of the circuit for fault injection and generating appropriate fault injectors; all the procedures are automated; this technique is very versatile and in a class by itself – it outperforms all known solutions for simulated fault injection at the gate-level;

- novel 64/32-bit RISC core architecture (including novel instruction set), presented in [Sim13] and Appendix A; the core is simple and flexible, with novel interrupt and virtual memory mechanisms;

- an 8-core framed multiprocessor based on the 32-bit RISC core is built according to the thesis proposal, for the purposes of evaluation of the framework; publication [SKK14b] presents the general concept and architecture;

- novel platform for automated HW/SW co-verification, testing and simulation of (multi)processors [SKK12], used to verify and test the design of the core and the multiprocessor (see Section 5.5); an assertion-based verification is carried out during software execution; additionally, various execution logs are produced; the instruction set simulation logs are compared to the register-transfer and gate-level simulation logs, which indicates potential flaws; furthermore, the automated fault injection procedure is integrated into this platform, and later used to evaluate the framed multiprocessor;

- novel register-transfer level NMR system generator, presented in [SKK13b]; given the parameter N, and the top file of a single module, the NMR system generator builds an NMR system ready for synthesis.

A very important characteristic of the architectural framework is that it can be built using virtually any type of cores, including COTS cores. In this way, both hardware and software design is leveraged: hardware IP cores could be directly integrated; already developed software procedures could be used and extended (e.g., operating system procedures like task scheduling).

It is also interesting to note that the largest part of solutions for multiprocessor dependability mentioned in Chapter 2 (e.g, core or thread-level redundant execution, NMR, duplicate-and-compare, pair-and-spare, NMR with K spares, self-purging, NMROD, aging-based scheduling, etc.) could be implemented, or, emulated by the proposed framework. That is, the framework could be used as a test and evaluation platform for various fault-tolerant techniques, thus increasing its scientific and practical value.

At last, a feature of this framework is its scalability (regarding the number of cores), which is crucial for multiprocessor systems (see Subsection 3.4).

Chapter 3

Architectural multiprocessor framework

As outlined in Section 1.4, the thesis investigates a multiprocessor framework with several modes of operation that are dynamically changed according to the current application requirements regarding performance and fault tolerance. The framework tries to satisfy these requirements at the lowest possible aging rate and power consumption. The three basic operating modes are: de-stress, fault-tolerant and high-performance.

The general architecture of a Framed MultiProcessor (FMP) with K Framework Groups (FWGs) was given in Fig. 1.13 on page 25. A short notation for this system is FMP$(P_1, P_2, \ldots, P_K)$, where $P_i, i \in \{0, 1, \ldots, K\}$ is the number of modules in the i-th FWG. Similarly, the notation $M(i, j)$ is used to denote module j of group i.

Fig. 3.1 presents a more detailed look into a FWG, in which the modules are processor cores. Optionally, one or more levels of cache could be present. All inputs and outputs of the cores are connected through the Framework Controller (FC). The FC generates control signals like interrupt and reset for all PEs, thus enabling the Framework Middleware (FM) to control the entire system. Global resets could be also generated by the FC. Subsections 4.1 and 4.2 give an in-depth overview of the FC and FM, respectively.

On the other side, Fig. 3.2 shows a FWG in which the modules are memory blocks. The FC is slightly different only in respect to the connection of the signals. For example, the memory blocks could not be interrupted. Here too, interrupt lines have to be connected to the PEs in the system. Furthermore, note the reversal of directions of the data, address and control lines. With some exceptions, the FC does not care about the logical meaning of the module's signals. Of course, it is important whether the signal is a module input or output.

Thus, building FMPs as in Fig. 1.13 is straightforward. Furthermore, if direct connections are used in simple systems like in Example 2 on page 25, only one FC is sufficient!

Details of an FMP(4, 4) produced in IHP 130 nm technology are given in Section 4.5.

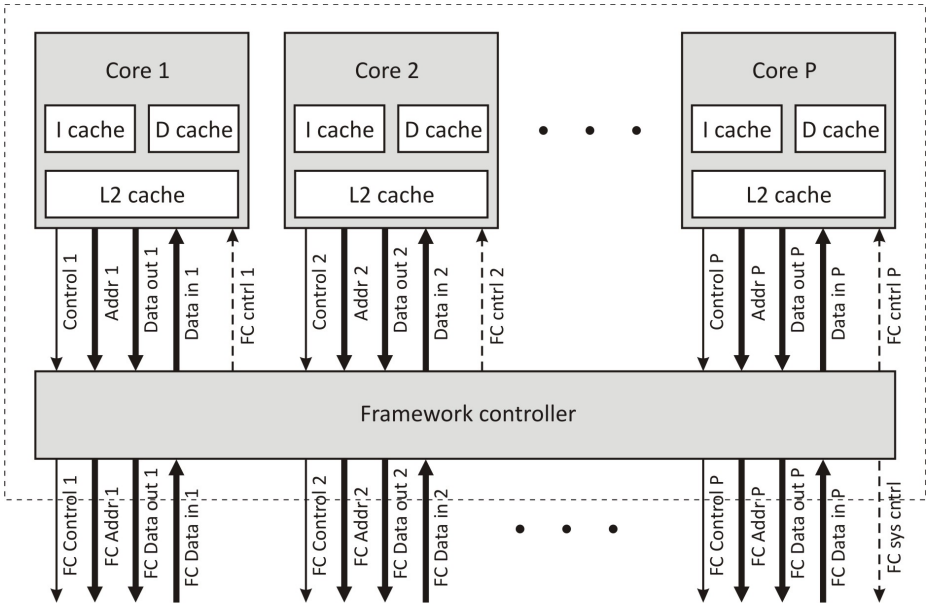


Figure 3.1: Processor cores arranged in a FWG. Inputs and outputs go through FC. Dashed lines indicate additional control signals generated by FC.

3.1 Operation in de-stress mode

If the current workload of the FMP requires only one or few modules operating, the rest of them could be inactive. *Inactive module* means that it is switched off the power supply or decoupled from the clock, in order to reduce power consumption and wear-out (aging effects). An *active module* performs its given function i.e., it is powered and clocked. In other words, the active module is under stress, while the inactive is being de-stressed. Almost complete de-stressing is achieved if the module is powered off, while decoupling only the clock partially de-stresses the module.

Furthermore, in order to de-stress the currently active modules, the workload could be transferred to the inactive modules (after activating them). Of course, the modules which were active up to now are deactivated. Doing this repetitively by some pattern may lead to longer system lifetime.

However, several issues and trade-offs have to be investigated. Firstly, the pattern and frequency of module gating i.e., (de)activation, as well as their (in)active periods may significantly affect aging and power consumption, but also performance. Too frequent module gating could make more damage in respect to aging and cause greater power consumption, than if the module is left active all the time. Moreover, performance overheads are increased. On the other hand, leaving the module active for too long, may negatively impact its lifetime.

Secondly, modules can be clock- or power-gated. Clock Gating (CG) could significantly reduce HCI aging effects and dynamic power consumption. However, static power consumption is not eliminated, while NBTI is even more pronounced. On the

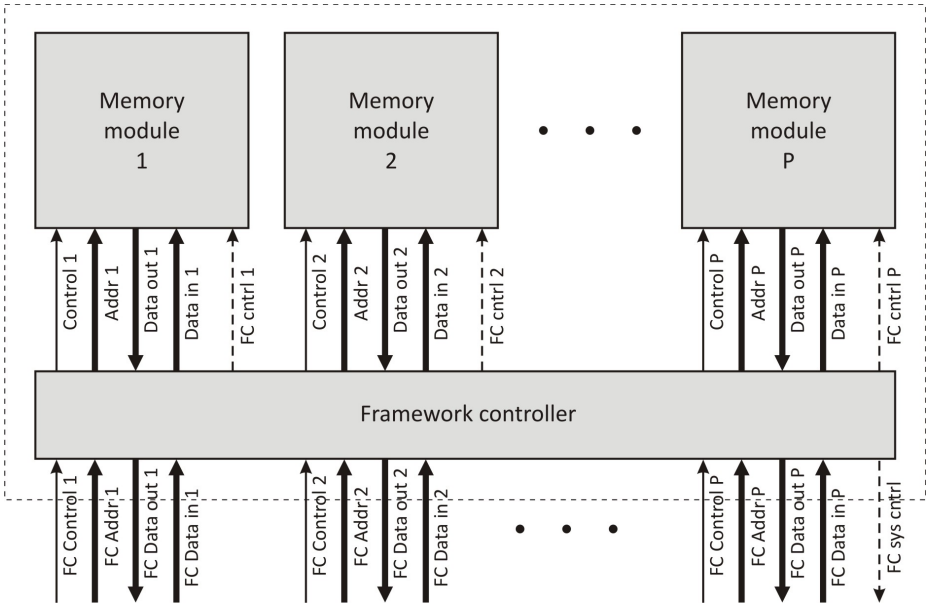


Figure 3.2: Memory modules arranged in a FWG. FC control signals (mainly interrupts) are connected to PEs.

other hand, Power Gating (PG) could lead to neglectable aging and power consumption, but the activation latency is significantly increased.

It is worth noting that de-stressing could be done beyond the borders of the framework groups. That is, the logical grouping of modules in the system does not play a role in de-stress mode. For example, execution of a set of tasks in a FMP(3, 3, 3) system with identical modules can begin at $M(1,1)$, continue with $M(1,2), M(1,3), M(2,1)$ and end at $M(2,2)$. This is suitably represented as:

$$\ldots \xrightarrow{T_0} \underline{M}(1,1) \xrightarrow{T_1} M(1,2) \xrightarrow{T_2} M(1,3) \xrightarrow{T_3} M(2,1) \xrightarrow{T_4} M(2,2) \xrightarrow{T_5} \ldots,$$

where T_0 to T_5 denote the active periods of the modules. The periods over the arrow refer to the modules that start the arrow, i.e., T_1 is the active period of $M(1,1)$, T_2 of $M(1,2)$, etc. $\underline{M}$ denotes the module that starts the execution of the observed set of tasks.

3.1.1 Module gating patterns

From a power consumption perspective, it does not matter if only a specific set of modules is mostly active, while other modules are mostly inactive. Alas, taking into account aging, makes a lot of difference. The goal of finding an appropriate gating pattern is to balance the load and equalize the aging between the modules in the system, which hopefully will prolong system lifetime. A pattern that equally distributes the load is Round-Robin (RR) with an active period T for all modules. Fig. 3.3 shows a RR gating pattern applied in an FMP(4) system. For simplicity of

presentation, the examples in the thesis consider mainly one active module. Patterns are easily extended for cases with up to $P_i - 1$ active modules, as Fig. 3.4 depicts.[1]

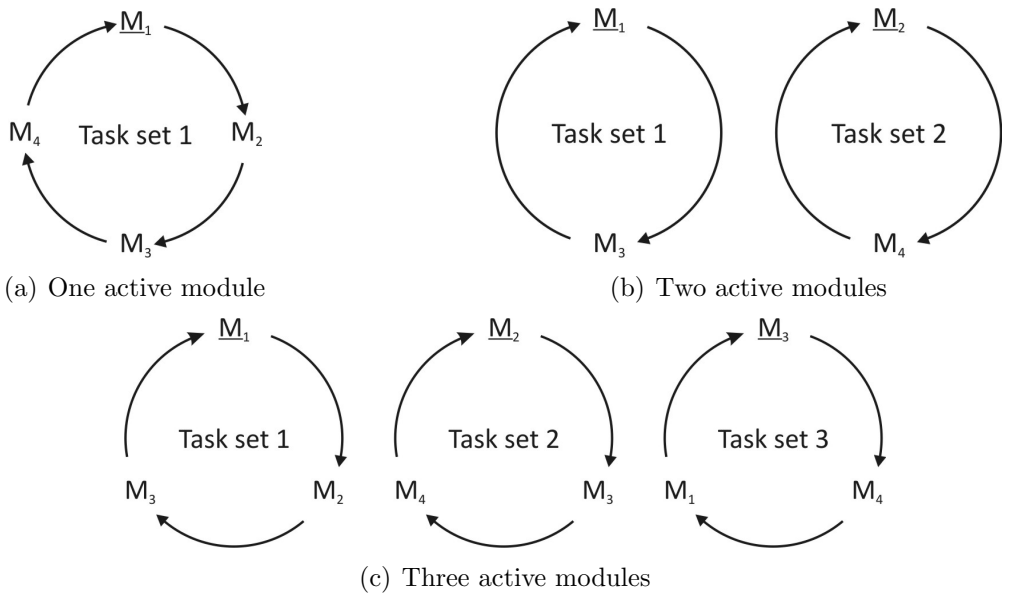


Figure 3.3: Round-robin gating pattern of an FMP(4). In systems with one FWG, modules are simply denoted as $M_j, j \in \{1, 2, \ldots, P_1\}$. Active periods T (not shown for clearness) are assumed in all transitions.

Fig. 3.3(b) and Fig. 3.3(c) do not clearly illustrate which modules are currently active and which are not. A more convenient presentation for that purpose is given by Fig. 3.4, showing when and which of the modules are active.

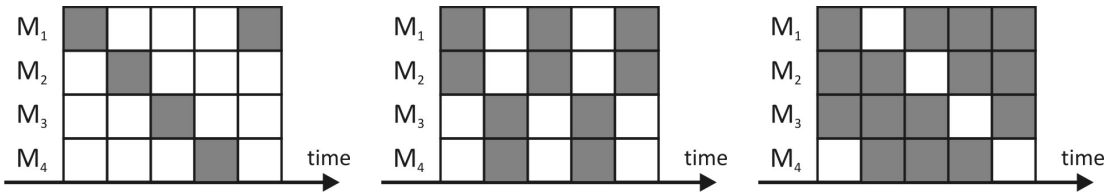


Figure 3.4: RR gating in time. Grey and white rectangles denote active and inactive modules, respectively. Since all transitions assume an active period T, each rectangle is T time units wide.

Nonetheless, an RR pattern with equal T for all modules could equalize aging only in a theoretical or perfect system. Factors like process variations may introduce "different ages" of the modules even before the system begins to function. This becomes worse if the modules are spanned across separate dies, ICs, or sub-systems.

[1]Having P_i active modules in each FWG does not de-stress the system.

A possible solution is to assign different active periods T for different modules, i.e., weighted RR. In order to do so, one has to estimate the age of the modules somehow.

Feedback on aging may be obtained by counting errors (see Subsection 2.2.2), using error detection circuits or schemes, like in [87]. As said, this is not very appropriate since errors are frequently induced by SEEs. Alternatively, one can rely on aging sensors or monitors like the ones presented in [117, 122, 79, 17]. One of the novel proposal of this thesis is a low-complexity, all-digital, self-calibrating, integrated circuit aging monitor [SKK11b], described here in Subsection 4.1.2. It is used as a mean to find the relative age of all modules in the multiprocessor, and based on this information, construct the gating pattern.

Youngest First RR (YFRR) could be a possible strategy to equalize the age of the modules and increase system lifetime. That is, the youngest modules are active until they equalize their age to another module. If several modules have the same (youngest) age, they are RR gated, while older modules are left inactive. Proceeding in this way, the system ends up using RR gating with all modules. Consider Example 7.

Example 7. *Let the modules M_1, M_2, M_3 and M_4 in a FMP(4) system are at the ages of 2, 3, 4 and 5, respectively. Assuming one active module at a time, a YFRR pattern will hold only M_1 active until it reaches the age of 3. Afterwards, M_1 and M_2 are interchangeably gated in a RR fashion, until they reach the age of 4. Similarly, M_1, M_2 and M_3 are gated in a RR fashion until they reach the age of 5. Then, all four modules are gated in a RR fashion. Fig. 3.5 graphs the YFRR gating pattern.*

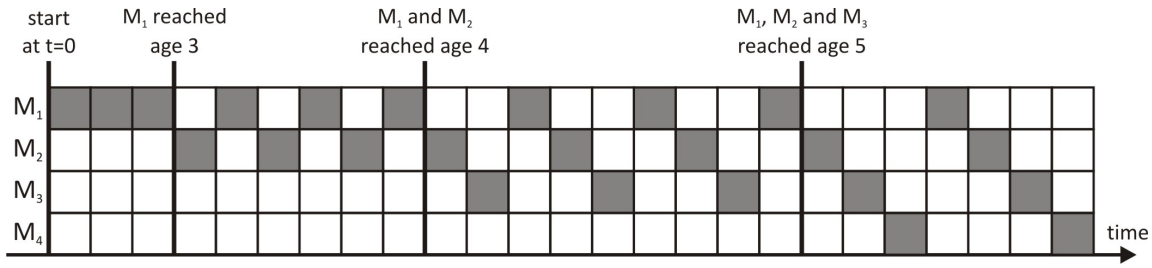


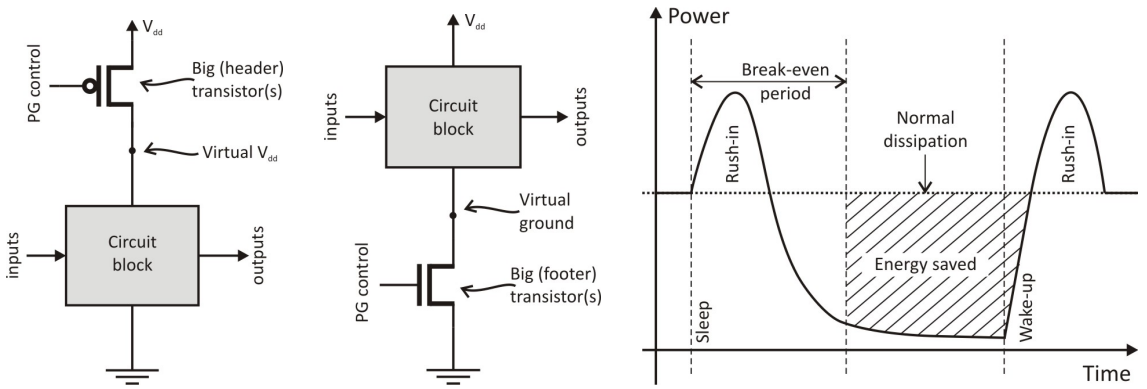
Figure 3.5: Youngest first RR module gating times

Note that in a YFRR pattern it is not necessary (although possible) to calculate special active periods for the modules, i.e., an active period T is always assumed.

3.1.2 Clock vs. power gating

Several trade-offs drive the choice between clock and power gating. On-chip PG is implemented by big header or footer transistors which control whether a certain block is powered or not (see Fig. 3.6(a) and 3.6(b)). Since the block is connected to V_{dd} or ground through these transistors i.e., to a virtual ground or virtual V_{dd}, there will be still some small current leakage causing static power dissipation when the block is powered off. Much more significant phenomena are the rush currents that flow during

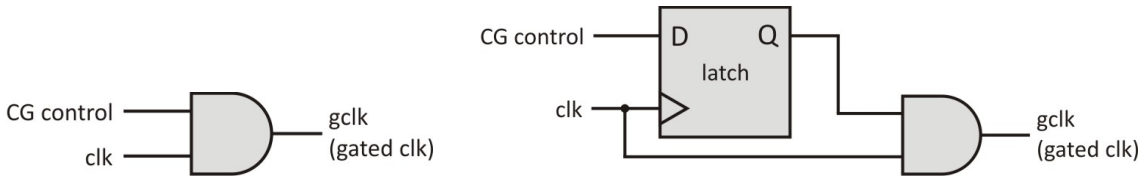
a limited time period after the block is switched on/off (see Fig. 3.6(c)). Thus, the off period of the block should be at least equal to the break-even period [71], in order to guarantee that the circuit will at least not increase the power consumption. Of course, these rush currents have negative impact on aging, e.g., they can significantly speed-up HCI.



(a) Using headers (b) Using footers (c) Dissipation, savings, break-even

Figure 3.6: Power gating

On the other hand, CG is implemented by logically "anding" the clock signal with the control signal (see Fig. 3.7(a)), usually with an additional latch that eliminates the glitches on the control signal (see Fig. 3.7(b)). A variety of CG circuits are given in [60]. However, CG does not eliminate static power dissipation (leakage), which in modern nanotechnologies is greater than dynamic power dissipation. Furthermore, when the clock is off, all negatively biased PMOS and positively biased NMOS transistors are more prone to the BTI effect, since now the circuit does not change states, i.e., the transistors are constantly under the same bias. HCI, on the other side, would be lowered since no significant currents flow through the transistors (except leakage).



(a) CG control anded with clk (b) CG control latched and anded with clk

Figure 3.7: Clock gating

Another aspect to take into account is the activation time, or, **activation latency**, which is directly related to performance. For instance, a large power-gated module has to be (de)activated part by part, in order to limit the rush currents that would otherwise damage the circuit. This is done using several headers or footers which are switched consecutively with some time distance. Depending on the module size, dozens of clock cycles may be needed for wake-up/go-to-sleep. Even worse,

consider the case where entire cores with L1 caches are subject to PG. The caches (at least instruction cache) have to be refilled on each core wake up, which depending on the cache type and block size, as well as memory connection and organization could take up to several thousand cycles. On the contrary, CG can bring the core back to function in one clock cycle, since the caches are powered and have not changed state. However, in the case of PG, clever schemes could be implemented: module activation could be done in parallel with another activity. For example, an FMP operating in de-stress mode could activate core X to take over the job of core Y in advance, just before the take-over, so there is no (or minimal) performance penalty.

In the proposed framework, both PG and CG are provided. The decision which one to use is left to the application layer.

3.1.3 Selecting an optimal (in)active period

Subsections 3.1.1 and 3.1.2 already discussed most of the aspects that direct the choice of the active period T. To summarize, the factors that should be taken into account when deciding on the active periods are:

- mission time,
- number of modules,
- number of concurrently active modules required,
- age of modules,
- gating pattern,
- gating type – clock or power.

As said, each module could have a special active period, assigned in accordance to its age. Alternatively, the system may opt for the YFRR pattern where each module could have the same active period. In both cases, the active period could be dynamically calculated and adapted according to the feedback on aging. For long-life systems, e.g., satellite on-board multiprocessor with a 10 year mission, active periods in the order of days, weeks, or even months could be selected, depending on the number of cores. Of course, switching the operating mode disturbs the de-stress (gating) pattern and the (in)active module times.

3.2 Operation in fault-tolerant mode

Fault-tolerant system design is driven by many factors such as the expected error rate, performance demands, power restrictions, as well as the available area, i.e., the cost of the system. As elaborated in Chapter 2, fault-tolerant mechanisms require redundancy of some form (information, time or space).

The multiprocessor framework in this work is based on coarse-grained NMR mechanism, where coarse structures like cores with or without caches, memory blocks, etc., are the "redundant" modules in the NMR system. A special programmable NMR voter (see Subsection 4.1.4) is used to dynamically form the NMR system from an arbitrary set of identical modules, and arbitrary N. In each clock cycle, the voter selects the majority output and takes additional actions, if needed.

The following considerations led to selection of these mechanisms in fault-tolerant mode. As outlined in Subsection 1.3.2, in order to increase performance, multiprocessors exploit parallelism in programs (and between programs) using multiple PEs.[2] When a PE is idle, it could be alternatively viewed as a redundant component – a requirement for fault tolerance. Thus, in a special, fault-tolerant operating mode, these PEs could re-execute jobs of other cores in order to confirm the results and increase reliability.

Using entire multiprocessor cores as "redundant" modules in a NMR system has several advantages. Firstly, any type of core (including already verified off-the-shelf IPs) could be used to build the multiprocessor, without any design changes. This is advantageous, as said, since both hardware and software design are leveraged. Secondly, the error coverage is greater in comparison to finer-grained solutions. Thirdly, additional fault-tolerant mechanisms are not needed. For example, if the L1 cache is part of the "redundant" module, it does not even have to include parity checks. Lastly, an NMR system (with $N > 2$) masks possible errors in each clock cycle, which is appropriate for timing-critical operations – the core performance is not affected at all. Furthermore, using the programmable voters, NMR systems could be dynamically formed, i.e., NMROD with arbitrary combination of N identical modules. (Static NMR is even more straightforward.) Actually, a great deal of fault-tolerant mechanisms described in Chapter 2, could be implemented by the proposed framework.

However, N has an upper bound due to practical limitations. As shown in Section 4.4.3 and [SHKK12], both area and propagation delay of the programmable voter scale quadratically with N. Because of these limitations, and with the purpose to build a scalable and orthogonal multiprocessor architecture, FWGs have to be defined. A FWG is a set of P identical modules in which NMR systems could be formed, where $1 \leq N \leq P$. On the other hand, redundant execution (for applications that are not timing-critical) by two or more cores like in [38], could be easily implemented using any set of cores, regardless of the FWG boundaries.

At the end, note that the "redundant" modules are not redundant in other modes of operation. E.g., high-performance mode uses the multiprocessor cores in exactly the same manner as any regular multiprocessor.

3.2.1 NMR system formation

Before elaborating how NMR systems are formed inside a FWG, one of the basic questions to be investigated is how many cores should a FWG contain, i.e., what is the optimal number P. Increasing P will maybe offer higher fault tolerance. On the

[2] On the other side, multiple memory banks are used to increase system throughput.

other side, the complexity, area, performance and power consumption overheads of the framework controller may outweigh the gains in fault tolerance.

As discussed, if only permanent faults are considered, a TMR system outperforms 4MR and 5MR (see Fig. 2.8), and is only slightly worse than 6MR. Several experiments were conducted in order to evaluate the behaviour under temporary faults. Section 6.2 details these experiments and their results, as well as results regarding the performance overheads when state recovery is engaged. Furthermore, Section 4.4.3 and [SHKK12] present an in-depth evaluation of the programmable NMR voters which are the main building blocks of the framework controllers. The insight into the trade-offs helps choosing P when building an FMP. However, for the purposes of this Chapter, $P = 4$ is mainly assumed, that is, FMP(4).

In a FMP(4), the framework controller should enable dynamically forming DMR, TMR and 4MR systems, using any combination of modules. Furthermore, two DMR systems at a time should be also possible. Fig. 3.8 shows all NMR configurations in a FMP(4).

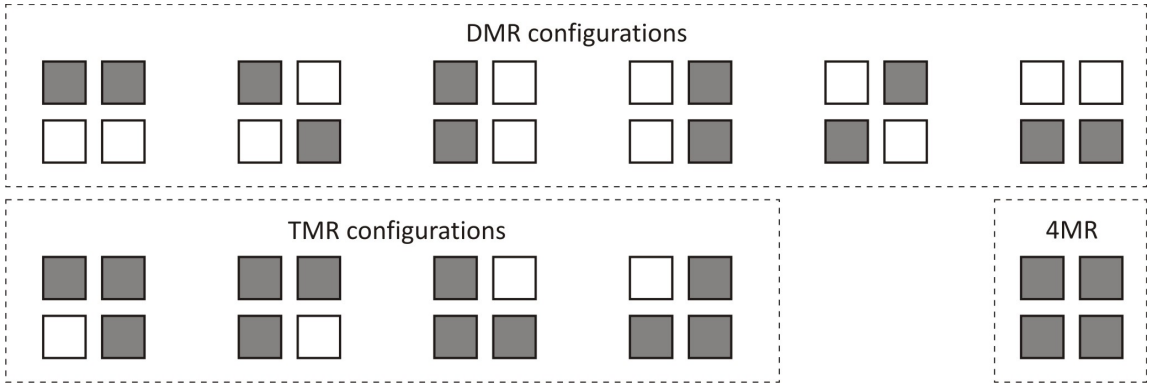


Figure 3.8: Possible NMR configurations in FMP(4)

The framework controller should be able to restructure the FWG in several clock cycles, e.g., go to TMR from DMR, or, change the combination of modules that form the TMR. In other words, transitions from a FWG configuration to another FWG configuration should be dynamically made upon a system or application request in just a few cycles.

3.2.2 State recovery

Modules in an NMR system could get into non-consistent states. For instance, if a fault in one or more cores in the FMP occurs, their state may no longer be the same as the state of the correct ones. Schemes for efficient state recovery of the erroneous modules in the system have to be investigated.

Several proposals for state recovery in NMR systems are given in [19]. Mechanisms of state recovery could be various: simply resetting all the modules that belong to the NMR system and restarting the computation, CRR, or copying the state from the non-faulty cores to the faulty ones.

Different solutions for state recovery impose different trade-offs on efficiency, performance overheads and power consumption. Besides quickly detecting the erroneous state, fast state recovery schemes require that the framework controller is able to pinpoint the erroneous module(s), as well as determine the number of erroneous modules. As will be shown, the programmable NMR voters have an Input State Descriptor (ISD) that outputs this information which is later used by the framework controller. Furthermore, the framework controller should be able to take the appropriate action efficiently: reset the module(s), or, invoke interrupts and state recovery handlers of one or more modules.

An intelligent state recovery scheme of a TMR system formed inside a FMP(4), where the fourth module is not needed and therefore deactivated, is the following. Let an error occur in one of the modules in the TMR system. The inactive module is then activated and included into the system. Operation can continue as long as the correct DMR system does not encounter errors. The erroneous module is deactivated (e.g., switched off the power supply or clock, or simply reset and not used). A special procedure, similar to a context-switch, is used to transfer the correct state from the error-free modules to the newly activated core. That is, the error-free cores save their state, and all three modules now forming the new TMR system resynchronize by reloading the saved state. This scheme is depicted in Fig. 3.9. Of course, if the error is found to be transient, the erroneous module itself could be brought back to a correct state instead of waking another module. Subsection 3.2.3 reveals how the framework controller determines whether the error is transient or permanent.

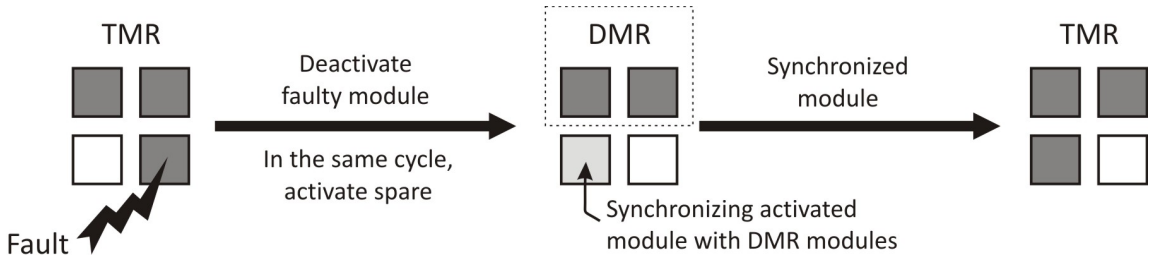


Figure 3.9: State recovery of a TMR in FMP(4)

If the modules are memory blocks instead of PEs, a simple rewrite of the erroneous word would be enough. The rewrite operation could be done by a PE in the system, or, alternatively, the framework controller could scrub that memory location without interrupting any PE – the correct data is present at the outputs of the non-erroneous memory blocks in the FWG.

An advantage of this state recovery scheme is that it de-stresses the module in which a fault was diagnosed, which is helpful in some cases (e.g., by NBTI-induced faults).

3.2.3 Fault classification

A simple scheme of distinguishing transient from permanent faults is counting errors in each module, during a specified *time period* (see Fig. 3.10). If the number of module

errors overcomes a predefined *error threshold* in this time period, a permanent fault is reported, else the error count for that module is reset to zero. Thus, the framework controller could decide to further use the module or to shut it down. Afterwards, the mechanism is initiated again and the procedures are repeated cyclically.

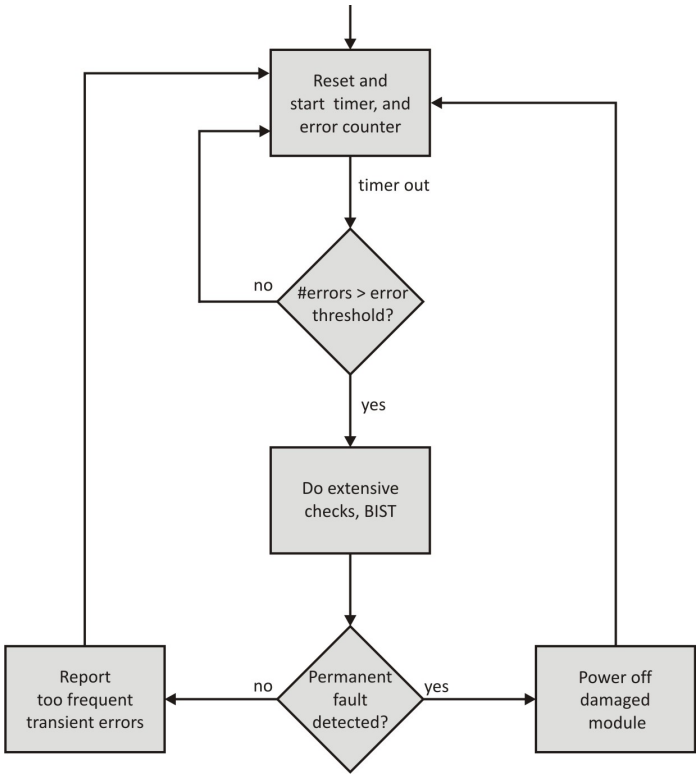


Figure 3.10: Fault classification scheme

The expected fault rate is the main guide when determining the time period and the error threshold for a specific system and environment. Fault injection could be used to find the optimal values of these parameters.

Before shutting a module down, more extensive checks could be initiated in order to confirm that the module is permanently damaged. Built-in self tests, or other diagnostic procedures are suitable for this purpose. For memory blocks, write-in and read-out of predefined patterns at each memory location could be performed.

Summed up, in order to efficiently support fault classification, the framework controller should additionally include error counters for each module in the FWG.

3.2.4 Framework fault tolerance

If a fault occurs in one of the modules operating in fault-tolerant mode, it will be masked by the voter. Nevertheless, if a fault occurs in the framework controller, the system may fail. The controller, which in this case is a single point of failure, has significant functions that drive the entire operation of the system. A potential fault could induce fatal SEFIs.

Therefore, a special attention to the framework controller's reliability should be paid. Subsections 4.1.4 and 4.4.1, as well as [SHKK12] describe the self-check capability of the programmable NMR voters.

Furthermore, it is worth noting that the framework controller is a relatively small circuitry compared to the rest of the system. For example, the synthesis results of the FMP(4,4) implementation presented in Section 4.5 show that the framework controllers area is 1,47% of the entire FMP. The voters occupy 27,2% of the framework controllers, or 0,4% of the FMP. Thus, approximately 1% of the logic should be additionally protected. The low area figure allows implementing this logic with larger and/or radiation-hardened cells, but also circuit-level solutions such as elaborate Error Detection and Correction (EDAC) codes.

3.3 Operation in high-performance mode

Previously was stated that a FMP operating in high-performance mode is similar in operation to any regular multiprocessor. Nonetheless, there are some differences. Firstly, task mapping and scheduling could be lifetime-aware (see Subsection 4.3.2), based on the aging information supplied by the aging monitors described in Subsection 4.1.2.

Secondly, the framework controller could be used to switch off the clock or power supply of the unneeded modules. That is, if the application requires parallel processing using M-out-of-N modules in the multiprocessor, the "oldest" $(N - M)$ modules could be deactivated. However, this is not the same behaviour as in de-stress mode, in which module gating patterns automatically transfer the workload to "fresh", inactive modules and de-stress the active ones. High-performance mode does not involve module gating patterns in order to avoid performance overheads. Though, it does involve simple, dynamic, on-demand (de)activation of modules.

3.4 Scalability

The scalability of the proposed concept has several aspects which are tightly related to the number of modules in the system. A special investigation should answer the question "how easy" the system is scaled, i.e., how performance, fault tolerance, lifetime, power consumption and cost are affected by varying the number of modules in the system. Of course, the type of the interconnection network directly answers many of these questions.

Moreover, besides scalability, the network should enable fault-tolerant interconnections of the modules, since one of the objectives of this thesis is fault-tolerant multiprocessor operation. Mesh, cube and k-ary n-cube networks are possible candidates because besides scalability and redundant links, they enable simple routing protocols. However, proposing an appropriate interconnection network topology and routing (switching) mechanisms for the multiprocessor framework is planned as a future work.

Chapter 4

Implementation

Chapter 3 explained the three operating modes (de-stress, fault-tolerant and high-performance) and stressed what the framework should contain, in general. This Chapter explains the implementation in great details. Firstly, the layers of the system (see Fig. 1.12 on page 25) are described in Sections 4.1, 4.2 and 4.3. Section 4.4 elaborates the design method for one of the most crucial elements of the thesis proposal – the programmable NMR voters. At the end of the Chapter, Section 4.5 presents the framed multiprocessor FMP(4, 4) that was built to explore the proposed concept.

4.1 Framework controller

Fig. 4.1 shows the block diagram of the framework controller (FC).

Before explaining the functions of the FC, let's first see how the component is accessed, i.e., how the PEs write and read the FC registers. All inputs and outputs of all modules in the FWG go through the FC. It is therefore enough to indicate the positions of the address, data and read/write control lines. The FC monitors the Output Multiplexing Logic (OML) inputs whether some of the modules selects the FC and issues a write or read command to a valid address of a register. It reacts appropriately by writing the data to the selected register, or, places the contents of the register at the corresponding data lines when read operation is requested.

Table 4.1 shows the complete register set of the FC. Note that some of the registers are omitted in Fig. 4.1 with the purpose not to overload the block diagram.

However, in a FWG of memory blocks, the read/write operations have still to be done by the PEs in the system. In this case, the register write-in and read-out directions should be reversed (see Fig. 4.1). Furthermore, the address, data and control signals now come from the Input Multiplexing Logic (IML). This Section mainly assumes a FC for a FWG of PEs, although all principles apply to a FC for a FWG of memory blocks too.

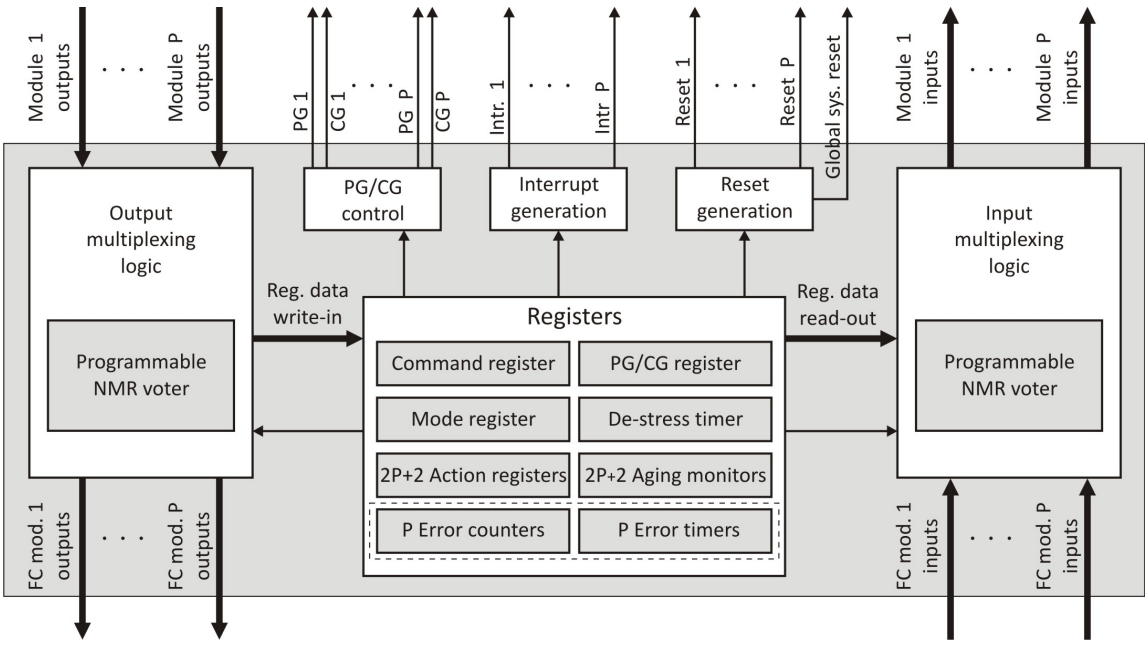


Figure 4.1: Framework controller. A FWG with P modules assumed. Thin arrow lines denote control flow. Thick arrow lines denote data flow.

4.1.1 PG/CG control and de-stressing support

The PG/CG control block contains CG cells as in Fig. 3.7, driving the clock of each module in the FWG. Furthermore, it drives the control signals of the PG header/footer transistors (see Fig. 3.6), thus controlling which module is powered on/off.

The framework middleware simply sets/resets the appropriate bits in the PG/CG register for the module(s) that have to be power-/clock-gated.

Note that if DVFS is used, this block would contain additional power and frequency control functions.

In de-stress mode, the de-stress timer is set to hold the active period T. On a timer-out event, the FC interrupts all active modules. The middleware handler takes control and (de)activates the modules by writing the PG/CG register, according to the information from the aging monitors and the gating pattern. The de-stress timer is run again, and the activated module(s) resume the work of the previously active module(s), after returning from the interrupt handler. Of course, the handler previously writes all process states in memory – similarly to a context switch procedure.

4.1.2 Aging monitors

The introductory Section 1.2.1 explains the aging effects in more details, while Section 1.3.1 shows the impacts of technology scaling on these effects. As said, the most dominant effects in CMOS technology are NBTI and HCI, which degrade the gate-oxide layers in the IC. In the thesis, their cumulative effect is used to construct gate-

Table 4.1: FC registers

Register	Description
Command	Basic commands for tweaking operation
PG/CG	PG/CG of modules in the FWG
Mode	Set NMR groups
De-stress timer	Timer for the de-stress period
2P+2 Action registers	Define actions for the formed NMR groups
2P+2 Aging monitors	HCI and NBTI aging monitors for each module and FC
P Error counters	Counting errors in NMR groups
P Error timers	Used to implement fault classification
Last action	Status and last taken action for a NMR group
Interrupt status	Pin-point and acknowledge generated interrupts
Interrupt mask	Enable/disable specific interrupts
Predefined outputs	Drive NMR outputs on voter error
Inactive outputs	Drive outputs when system environment is not NMR

oxide aging monitors. The YFRR core gating pattern, as well as the lifetime-aware scheduling and task mapping, use the information supplied by the aging monitors.

Fig. 1.6 on page 11 could help explaining the cumulative effect of NBTI and HCI in a simplified manner, using the classical CMOS inverter.

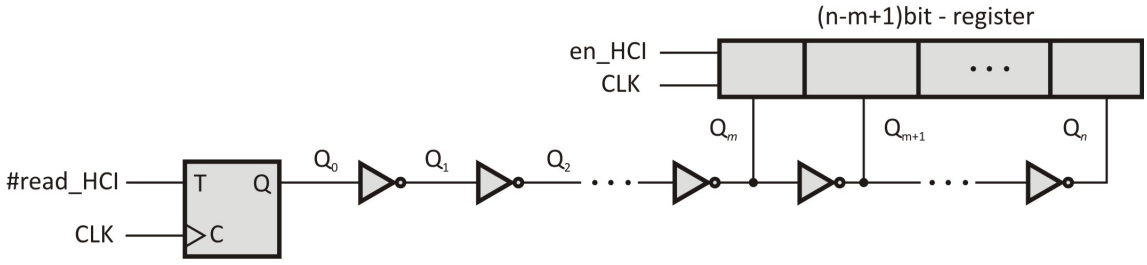
NBTI occurs when the inverter is in a stable state and its output is logic 0 ($\sim$ 0 V). Since the input is at logic 1 (around Vdd), the PMOS transistor is off and negatively biased. The negative bias triggers an electrochemical process – trapping and releasing carriers from the channel. Trapped carriers reduce the mobility of the channel carriers which in effect shifts the threshold voltage of the transistor. In older technologies NBTI affects only PMOS transistors. However, in newer ones, the same effect (PBTI) is observed in NMOS transistors too. Luckily, this process is reversible to some extent. If the inverter is not powered, most of the carriers are released back to the channel.

HCI occurs when the inverter changes the state i.e., transistors switch on/off. On a switch, both transistors conduct during a small time interval, in which a relatively large current flows from Vdd to gnd. Thus, some of the "hot carriers" are able to get sufficient kinetic energy to pass the Si/SiO interface and enter the forbidden oxide layer. HCI affects both NMOS and PMOS transistors. Unfortunately, HCI degradation is irreversible. The trapped carriers cannot be released back to the channel, eventually causing oxide breakdown – the carriers form a conductive path through the oxide.

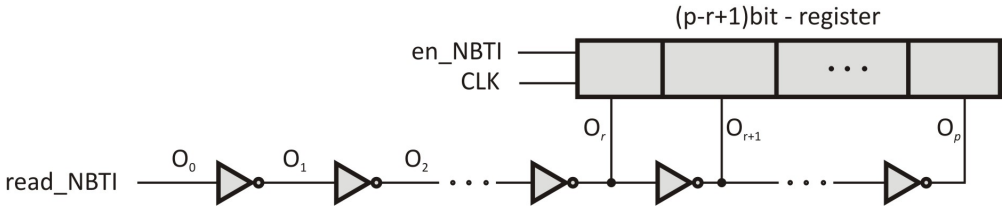
Thus, both NBTI and HCI increase the threshold voltage of the transistors, i.e., increase their input-to-output delay. The cumulative effect is lowered IC performance (frequency of operation). Ultimately, the transistors may get permanently damaged.

This cumulative effect is used to build NBTI and HCI aging monitors, proposed in [SKK11b]. In short, both of them are based on inverter chains, with the HCI chain

being switched on each clock cycle, while the NBTI monitor is never switched (except during age read-out). Fig. 4.2 shows both aging monitors.



(a) HCI monitor



(b) NBTI monitor

Figure 4.2: Gate-oxide aging monitors

Gate-delay due to gate-oxide aging effects will eventually obstruct the signal to propagate through all the inverters in the chain. As the circuit gets older, more inverters at the end will not feel the signal change in one clock period. Thus, an information that shows the level of circuit aging could be extracted.

For example, in a monitor with 16 inverters and an 8-bit register the "age code" of 01010101 means that the signal successfully propagated through all 16 inverters. A code of 01010100, on the other hand, indicates that the signal did not propagate through the last inverter in the chain. Similarly, 01010110 indicates that the last two inverters did not feel the signal change, etc.

The length of the chain should be constructed according to the target frequency of the modules. For example, the implemented FMP(4, 4) (see Section 4.5) has 320 inverters at 50 MHz. Though, the register width could be smaller. For practical reasons, it is handy to be an integer multiple of the data bus width of the module (e.g., the width of these registers in the implemented FMP is 64-bit).

Although the aging monitors logically belong to the FC, the PG/CG functions affect them exactly as they affect the corresponding modules. That is, if a module is power- or clock-gated, the corresponding monitor is also power-/clock-gated in the same time, and in the same manner. Extraction of the age information using the monitors has to be done in two steps: firstly, activate age read-out by setting an appropriate bit in the command register, and secondly, read the age code in the monitor's register. However, factors like temperature may affect the results and lead to deviations. In order to get a clearer picture of the age code, several read-outs and averaging the results are recommended.

Note that there are one HCI and one NBTI monitor per module. Thus, in a FWG with P modules, 2P aging registers have to be read. Additional pair of an HCI and

an NBTI monitor which is constantly powered and clocked observes the aging of the FC itself.

4.1.3 NMR system formation

The NMR formation inside a FWG is done simply by programming the mode register shown in Fig. 4.3.

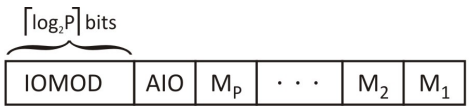


Figure 4.3: Mode register. All fields are one bit wide, except IOMOD which is $\lceil \log_2 P \rceil$ bits wide.

If bit M_i is set, the corresponding module is part of the NMR group, otherwise it is an independently operated module. Minimum two bits have to be set in order to form at least a DMR, otherwise no NMR group is formed. Setting all bits makes the entire FWG an NMR system consisting of P redundant modules. From outside, an NMR group is viewed as a single module (as in any NMR system). Of course, if the module is powered or clocked off it is not considered as a member of the NMR group, in which case the M_i bit is a "don't care" bit.

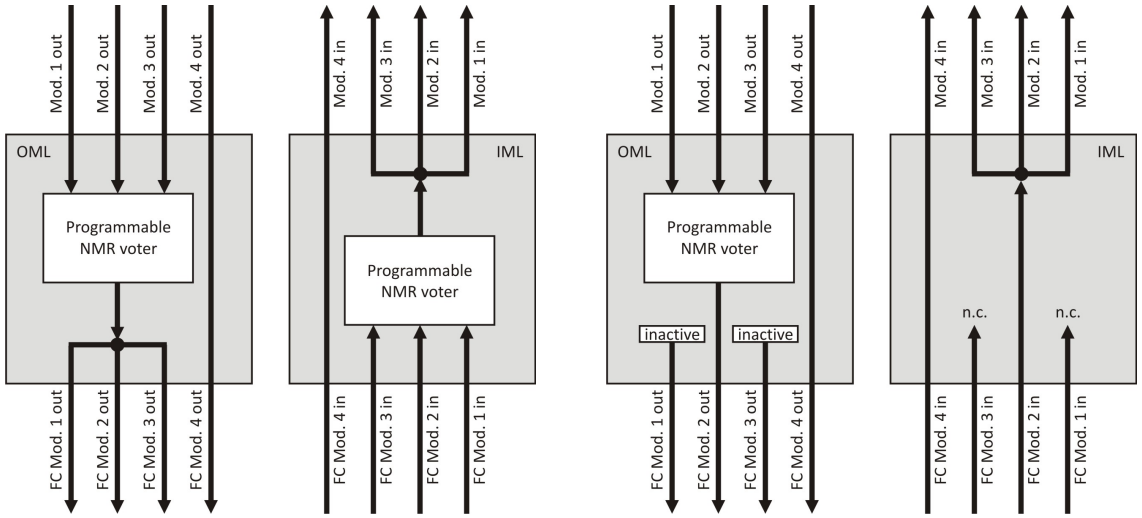
Setting the mode register configures the OML and IML, and the programmable NMR voters inside them.

Example 8. *Let the PEs 1, 2 and 3 form a TMR in a FMP(4), and let PE 4 be standalone. M_1, M_2 and M_3 have to be set by writing the mode register. The outside world sees a two-core system consisting of PE 4 and the TMR group of PEs. One thing left to convey to the FC is the environment in which this FMP is supposed to operate. If the AIO (All I/O) bit is set, the FC considers that the TMR system is extended outside the FWG. E.g., the inputs/outputs corresponding to PEs 1, 2 and 3 are connected to redundant memory blocks. Therefore, it configures the system as in Fig. 4.4(a).*

On the other side, if AIO is reset, the FC considers that the formed TMR group of PEs is connected to a single memory block, and configures the system as in Fig. 4.4(b). The communication between the TMR group and the external memory block uses the lines of the PE specified in IOMOD (IO module). In this example, IOMOD is set to 2. Setting IOMOD to 4 will trigger the "false setup" bit (FSET) in the last action register since PE 4 is not a part of the NMR group (M_4 is reset). The "inactive outputs" register now drives all outputs of PEs that belong to the NMR group except the PE specified in IOMOD. The IML in this case, does not use the voter.

4.1.4 Programmable NMR voters

The characteristics of the NMR system are mostly determined by the type of the voter. One of the contributions of this thesis is the novel type of programmable NMR



(a) AIO is set (b) AIO is reset, IOMOD=2

Figure 4.4: I/O configurations of a TMR in FMP(4)

voters which are presented in this Subsection. A design method for the novel voters, and their evaluation regarding performance and area is given in Section 4.4, as well as in [SHKK12].

A few definitions follow. Assuming that there are P redundant modules, let the set of inputs of the NMR voter be $\mathcal{A} = \{x_0, x_1, \ldots, x_{P-1}\}$, and the voting output be y. The absolute difference between two input values x_i and x_j is $\delta_{ij} = |x_j - x_i|$. The easiest to implement, and actually the most used algorithm is the exact voting algorithm where δ_{ij} must be 0 in order to consider x_i and x_j equal. An inexact voting algorithm on the other hand, defines σ for which if $\delta_{ij} < \sigma$ then x_i and x_j are considered equal. The third type is approved voting where each input consists of a set (or range) of approved values. The voter in this case outputs the most appropriate set of input values.

A general M-out-of-N voter considers the voting successful if there are at least M equal inputs (of N inputs in total). However, if $M \leq N/2$ then ambiguous situations exist, in which 2 or more candidates could be legitimate outputs. For instance, let in a 2-out-of-4 system $x_0 = x_1 \neq x_2 = x_3$. Two values are legitimate candidates for the voting output – the voter hesitates what to choose: $y = x_0 = x_1$, or, $y = x_2 = x_3$?

Fig. 4.5 shows the interface of the 1-out-of-N, programmable NMR voter used in the thesis. This voter has such a capability that N could be dynamically set in the range $1 \leq N \leq P$, using any combination of redundant modules. The x_i inputs and the output y can be W-bits wide. The voting output y is always equal to x_i, where x_i belongs to the largest group of equal inputs. Furthermore, the voter gives a complete description of the situation at its inputs x_i i.e.,

- the number of inputs which differ from the determined output – *nr_diff* (or, equivalently, the number of inputs which are equal to the determined output – *nr_eq*),

- the signals e_i which signal whether x_i is equal to y, and
- the ambiguous situation signal – *amb*.

These additional outputs are denoted as the Inputs State Descriptor (ISD).

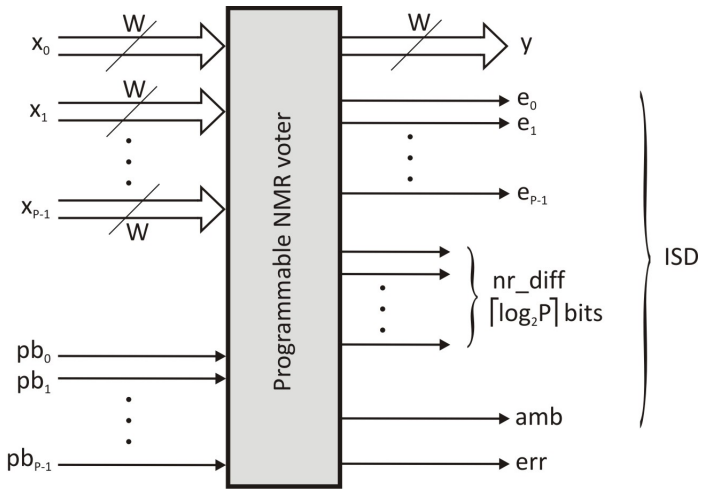


Figure 4.5: Programmable NMR voter with ISD and self-checks

Moreover, the voter does on-line self-checking of its operation. Inconsistencies are reported by asserting the *err* signal. Details of the self-check function are given in Subsection 4.4.1.

The programming is done in the following way. Each input x_i has an associated programming bit pb_i, which signals whether the input is to be considered for voting, i.e., whether x_i is an *active input*. Thus, in Example 8, the FC will configure $pb_0 = pb_1 = pb_2 = 1$ and $pb_3 = 0$ of both IML and OML voters. In other words, x_0, x_1 and x_2 are set as active inputs, while x_3 as inactive.

Thus, dynamical reconfiguration could be done. 1MR, 2MR,..., PMR systems with any combination of redundant modules could be formed. There are two exotic situations though: configuring all the inputs as inactive is considered illegal (y could be anything in 0MR); in a 1MR system on the other hand, the output y is always equal to the active input. For proper operation, at least one input should be defined as active.

Last characteristic but not least important is that the design of these voters is easy-scalable in respect to both P and W parameters (see Section 4.4).

4.1.5 Error handling – interrupt and reset generation

The ISD outputs of the voters enable precise definitions of actions. In each clock cycle, the FC knows whether all of the modules in the formed NMR group agree, or, which of the modules disagree, their total number, and whether an ambiguous situation exists. Fig. 4.6 shows the action registers which define the actions that should be taken on all possible events.

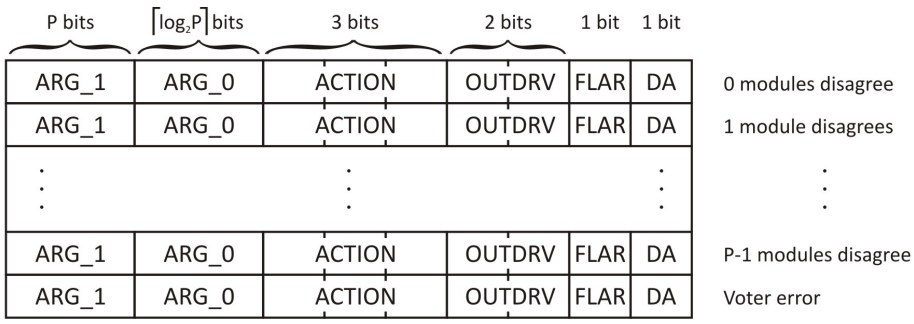


Figure 4.6: Action registers. Two packs of P+1 registers (one for IML, and one for OML) define what actions should be taken for each possible *nr_diff* output of the IML and OML voters, correspondingly. The "voter error" register defines the actions when the voter itself reports erroneous operation, and supersedes all other action registers on that event. If the formed NMR group has N modules where $N < P$, only the top N registers and the "voter error" register are valid.

The action registers could be programmed to reset or interrupt one or more modules, as well as invoke a global reset. This is necessary in order to bring all the NMR modules back to a consistent state, but also to enable uninterrupted operation in timing-critical procedures. Table 4.2 shows the actions that may be triggered according to the 3-bit ACTION field.

Table 4.2: Actions. Some of the actions use the P-bit ARG1 field that indicates the modules for which the actions apply.

Action code	**Action**
000	Raise interrupt to modules specified by ARG1
001	Raise interrupt to erroneous modules
010	Raise interrupt to non-erroneous modules
011	Not used. (Sets the "false setup" bit FSET.)
100	Reset modules specified by ARG1
101	Reset erroneous modules
110	Reset non-erroneous modules
111	General reset

If set, the DA (disable actions) and FLAR (freeze last action register) bits further set the eponymous bits in the command register (see Fig. 4.9), disabling successive actions (resets and interrupts), and updates of the last action register (see Fig. 4.7). The reasoning behind these bits is that once a module gets into erroneous state, it could trigger multiple interrupts and resets, which may prohibit efficient state recovery in some situations. Re-enabling actions and updates of the last action register are done by resetting these bits in the command register.

Besides definition of actions, it is necessary to drive the output of the NMR group in cases where one or more modules are erroneous. Normally, the voter output y is the most appropriate choice whenever there is a majority. However, when the voter operation is erroneous (reported by the *err* signal), or indecisive (reported by the *amb* signal), or when there is no majority ($nr\_diff \leq N/2$), a "more trusted" module could be set to drive the outputs of the NMR group. Alternatively, the "predefined outputs" register could be also configured for the same purpose. Table 4.3 shows the output drivers which can be selected by the OUTDRV field.

Table 4.3: Output driver selection. In the '10' case, the $\lceil \log_2 P \rceil$-bit ARG0 field is set to indicate the module which drives the output; note that the module does not have to belong to the formed NMR group (when $N < P$).

Outdrv code	**Output driver**
00	Voter drives NMR group output(s)
01	Not used. (Sets the "false setup" bit FSET.)
10	Module ARG0 drives output(s)
11	The "predefined outputs" register drives output(s)

As its name suggests, the read-only "last action" register shown in Fig. 4.7 records the last action that was taken (if the FLAR bit is not set)[1]. The interrupt handler can thus inspect what happened and make intelligent decisions.

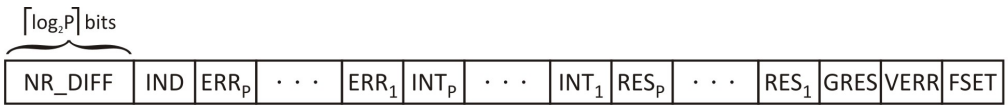


Figure 4.7: Last action register

The FSET bit is set if the FC configuration is false. The VERR bit indicates that a voter error occured. The GRES bit indicates that the last action was a global reset initiated by the FC. If the global reset comes from outside, the GRES bit is not set. RES_1 to RES_P show which of the modules were reset (if any). INT_1 to INT_P show which of the modules were interrupted (if any). ERR_1 to ERR_P show which of the modules erred – just by registering the e_i signals of the voter's ISD. The IND bit indicates whether the voter was indecisive by registering its *amb* output, while the NR_DIFF field registers the *nr_diff* output of the voter.

4.1.6 Other control and observation functions

When an NMR group is formed, each disagreement of a module output with the voter output is additionally noted by incrementing an error counter. In a FWG(P), the FC

[1] Actually, more appropriate name would be "last action and status" register. For short, only the "last action" part is used.

has P error counters in total. Furthermore, the FC supports the fault classification scheme discussed in Subsection 3.2.3. For that purpose, P error timers are provided to count the predefined time periods. If the module is not a member of the NMR group, or it is not powered or clocked, the corresponding error timer is inactive. Otherwise, writing a value greater than zero in the error timer register, starts the timer. On a timer-out event (when the error timer reaches zero), the corresponding module is interrupted.

The interrupt status register shown in Fig. 4.8 could be inspected to see the sources of the generated FC interrupts. Resetting the corresponding bits of this register tells the FC which of the generated interrupt(s) are handled. Table 4.4 describes the interrupt sources.

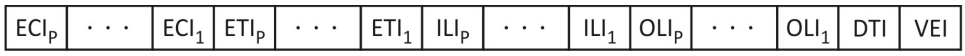


Figure 4.8: Interrupt status register

Equal in size, and with corresponding bit positions is the interrupt mask register which defines whether the interrupts are masked or not. Masked interrupts do not trigger events outside the FC.

Table 4.4: Interrupt sources.

Interrupt	**Description**
VEI	Voter error interrupt
DTI	De-stress timer interrupt (timer-out event)
OVI_1 – OVI_P	OML voter interrupt (i voter disagreements)
IVI_1 – IVI_P	IML voter interrupt (i voter disagreements)
ETI_1 – ETI_P	Error timer interrupt (timer-out of i-th timer(s))
ECI_1 – ECI_P	Error counter interrupt (the i-th counter(s) overflowed)

Finally, the write-only command register shown in Fig. 4.9 provides additional functions.

Figure 4.9: Command register

The DA and FLAR bits are explained in Subsection 4.1.5. The GR bit could be set to trigger global reset, while bits RM_1 to RM_P trigger resets to the corresponding modules in the FWG. Similarly, REC_1 to REC_P reset the error counters. AGE_1 to AGE_P initiate age read-out of the corresponding modules, while AGE_0 initiates age read-out of the FC itself. As previously explained, the age codes should be read from the corresponding aging monitors, after initiating age read-out in the command register.

4.2 Framework middleware

This Section details the software part of the framework – the framework middleware (FM). The FM is the middle layer in the framework (see Fig. 1.12 on page 25) which uses the services of the FC, and offers its services to the upper application layer.

The FM consists of two parts. The first part is a library of procedures that greatly simplify the communication with the FC, by hiding all details like addresses of registers and positions of various bits. For instance, simply invoking the procedure `powerOff(3)` will power off the module 3 in the FWG. Here, only the number of the module which should be switched off is supplied. The address of the PG/CG register as well as the bit position that controls the power state of module 3 are handled in the FM procedure – the application layer only calls the procedure with the appropriate argument. Furthermore, using powerful constructs of high-level languages, like enumeration types, the numbers of the modules could be replaced by symbolic names. Thus, calling the procedure would be e.g., `powerOff(BACKUP_CORE)`.

The second part of the framework are the interrupt handlers. Each PE in the system which could be interrupted by the FC should implement the main FC interrupt handler, or, handler dispatcher. The dispatcher should investigate the interrupt status register to see which FC interrupts are pending (see Fig. 4.8 and Table 4.4). If more than one interrupts are pending, the one with the highest priority is selected. In this Section it is assumed that the priority of FC interrupts is defined according to the bit positions in Fig. 4.8, i.e., VEI is the highest priority interrupt, while ECI_P is the lowest. However, each specific system could define its own FC interrupt handlers (and their priorities), which may deviate from the ones presented here.

Throughout the Section, a pseudo programming language (similar to C) is used to describe the FM procedures which are written in a fixed-width font. For instance, Fig. 4.10 shows the interrupt handler dispatcher.

```
while(ifIntrPending())                    //while pending interrupts
{
  int ISTAT = getIntrStatus();           //copy intr status to ISTAT
  int INTR = findHPIntr(ISTAT);         //find highest priority intr
  int HNDL_ADDR = intrHndlAddr(INTR);    //get intr handler address
  jmp HNDL_ADDR;                                  //jump to handler
  ackIntr(INTR);                        //acknowledge interrupt
}
return;                                  //return from dispatcher
```

Figure 4.10: Interrupt handler dispatcher. Here, `findHPIntr()` and `intrHndlAddr()` are not part of the FM. All other functions are explained later. An integer variable is defined by `int`, while a single bit boolean variable is defined by `bool`. `ifIntrPending()` for example, returns boolean.

4.2.1 Library of framework procedures

Table B.1 on page 151 in Appendix B lists the library of FM procedures. The application layer controls and observes the framework solely by invoking these procedures.

Note that set and get functions like `getPGCG()`, `setPGCG(int)`, `getMode()`, or `setMode(int)` should be used only for storing or retrieving the state of the FC. Otherwise, one has to know the exact bit positions of registers which is contrary to the purpose of this library.

Procedures related to action registers contain the word `Action`, and are actually aliases of "`OMLAction`" procedures. The completely equivalent "`IMLAction`" procedures in which only the corresponding register addresses are different, are not shown in Table B.1. Thus, for example, if one wants to get all actions of the OML, should either call `getActionsAll()` or `getOMLActionsAll()`; to get all actions of the IML, `getIMLActionsAll()` should be called. This is the case for each function that relates both to OML and IML.

Procedures related to PG/CG, de-stress mode and age monitoring

The most exotic function in the FM library is the `*int getAge(*int)` function since it involves two different types of registers – the command register and the aging monitors. For instance, in order to get the age of all modules and the FC in FMP(4), one should call `getAge(0,1,2,3,4)`. Alternatively, `*int getAgeAll()` could be invoked, which assumes all possible age codes, and therefore no arguments are needed. The procedure firstly initiates age read-out in the aging monitors by writing the corresponding AGE_0 to AGE_4 bits in the command register. Then, it reads all HCI and NBTI monitors in each module and the FC, and packs the information in the return `*int` array. Note that the output array has twice as many elements as the input array. The procedure is shown in Fig. 4.11.

For de-stress operation, e.g., implementation of the YFRR pattern, two additional registers are crucial besides the aging moniors – the de-stress timer and the PG/CG register. The de-stress timer is activated by simply writing a non-zero value to it, using the `setDeStressPeriod(int)` function, while `noDeStress()` deactivates it by writing a zero value. The function `ifCouldDeStress()` checks if there is at least one inactive module, so a de-stress pattern could be applied.

Procedures related to NMR system formation and error handling

Forming an NMR group is simply done by the `formNMR()` procedure, by specifying the module numbers that form the group as arguments. For example, `formNMR(2,4)` forms a DMR group of the modules 2 and 4. The procedure only writes the corresponding bits of the mode register. Of course, there are a dozen of other functions that ease mode setup.

Defining actions is a bit more trickier, because of the greater number of arguments that need to be set. The `setAction()` procedure has two versions which differ by the number of specified arguments. The first version has two arguments: the action register number (from 0 to P), and the contents of the register. Nevertheless, much

```
*int OUTARR getAge(*int INARR)
{
  int AGE_BITS = 0;                    //initialize all AGE_BITS to zero
  foreach INARR                              //loop through input array:
  {                                          //for each specified module
    AGE_BITS($_) = 1;                           //set corresponding bit
  }
  AGE_BITS = sl(AGE_BITS, 3+2P);          //shift left 3+2P positions
                                    //to reach AGE bits of command reg.
  store AGE_BITS, [CMR_ADDR];             //store in command register
  nop;                                //no operation, wait for age read-out
  *int HCIARR;                                        //declare HCI array
  *int NBTIARR;                                      //declare NBTI array
  int i = 0;
  foreach INARR
  {
    load HCIARR(i), [HCI_ADDR($_)];          //read $_-th HCI monitor
    load NBTIARR(i), [NBTI_ADDR($_)];       //read $_-th NBTI monitor
    i = i+1;
  }
  OUTARR = concatArrays(HCIARR, NBTIARR);  //concatenate HCI and NBTI
                                                    //arrays into OUTARR
  return OUTARR;
}
```

Figure 4.11: Age read-out procedure. As usual, P is the predefined constant of the number of modules in the FWG. `concatArray()` concatenates two or more arrays into a single one. $_ is a Perl-like variable that contains the current element of iteration of the loop. E.g., if INARR = (2, 4, 5), in the three iterations of `foreach INARR`, $_ will have the values of 2, 4 and 5, respectively. `load` and `store` read and write data from/to memory (or, memory-mapped components as the FC), where the contents in angle brackets denotes the memory address.

more handy is the second version shown in Fig. 4.12 which has seven arguments that specify each field of the selected action register.

In order to further ease the programming, enumeration types could be used to give symbolic names to numbers. For example,

```
OUTDRV_FIELD enum VOTER, MODULE=2, PREDEFINED;
ACTION_FIELD enum INTR, INTR_ERR, INTR_NO_ERR, RESET=4, RESET_ERR,
                  RESET_NO_ERR, RESET_GLOBAL;
```

set symbolic names to the corresponding values of Tables 4.2 and 4.3. Thus, in order to set an action upon an OML voter error that invokes a reset to modules 1 and 4,

```
void setAction(int NR_REG, bool DA, bool FLAR, int OUTDRV,
               int ACTION, int ARG0, *int ARG1)
{
  int LOG2P = roundUp(log2(P));              //the width of ARG0 field
  bit_array[2] OD = LSBits(OUTDRV, 2);       //get 2 LS bits of OUTDRV
  bit_array[3] ACT = LSBits(ACTION, 3);
  bit_array[LOG2P] ARG0_F = LSBits(ARG0, LOG2P);
  bit_array[P] ARG1_F = 0;
  foreach ARG1                               //loop through ARG1 array:
  {                                         //for each specified module
    ARG1_F($_) = 1;                            //set corresponding bit
  }
  int FVAL = concatBits(ARG1_F, ARG0_F, ACT, OD, FLAR, DA);
  store FVAL, [ACTION_ADDR(NR_REG)];         //store in action register
}
```

Figure 4.12: Set actions procedure. NR_REG is the number of action register (0 to P). All other arguments correspond to the fields of the action registers. Note that `bit_array[]` defines an array of bits. `LSBits()` returns the least significant bits of the integer argument. `concatBits()` concatenates the input argument bits and returns an integer.

where module 3 drives the outputs, and where DA and FLAR bits are set, one could invoke the procedure as follows:
`setAction(P, TRUE, TRUE, MODULE, RESET, 3, (1, 4))`.

4.2.2 Interrupt handlers

As already stressed, each system may define its own FC interrupt handlers according to the application needs. However, this Subsection reviews the handlers needed to support the framework functions like YFRR de-stressing, state recovery, fault classification, etc., that were described in previous Sections. Some of the handlers are specific though, and may largely differ in different applications. For example, the voter error interrupt (VEI) reports a serious flaw in the system – the voter of the formed NMR group itself is a subject of faults. Apparently, the most appropriate action would be reset all NMR modules, or, do a global reset instead of generating interrupts. Nonetheless, for diagnostic or other specific purposes, interrupts to modules that do not belong to the NMR group could be invoked.

YFRR software handler

The de-stress timer interrupt (DTI) handler is given in Fig. 4.13. It signals the timer-out event of the de-stress timer. If YFRR is employed, this means that now at least one active module should be inactivated. Furthermore, modules that were inactive

up to now, should be activated to take over the job of the modules that are going to be inactivated. At the end, the de-stress timer should be set again to countdown the same period. A short description of this handler follows.

The `*int getAgeModulesAll()` procedure is another alias of `getAge()`, which is similar to `getAgeAll()` except that the age of the FC itself is not returned.

The `*int youngestFirst(*int)` procedure is a simple sort procedure that just sorts the input array – the youngest modules come first (at lower indexes). It uses the `getAge()` procedure to investigate the age of the modules specified in the input array. It returns the sorted array.

The `bool ifDestressSwitcher(int)` procedure just checks if the input argument is the first active module in the array returned by `getActiveModules()`. Only one active module should inactivate all others (possibly itself too), as well as activate the youngest modules, otherwise a chaos may occur. Note that the `activate(*int)` procedure sets the corresponding bits in the PG and CG register. Thus, if the module is already active, this procedure has no effect.

Lastly, `saveProcessState(int)` and `loadProcessState(int)` save and load the process state of the module specified as argument. These procedures are similar to a context switch and heavily depend on the instruction set of the PE.

Now let's see how the entire system functions regarding DTI. As said, the de-stress timer interrupts all active modules in the FWG on a timer-out event. Thus, all active PEs will jump to the same instruction handler. For the purposes of simple synchronization, it is important to set high priority of DTI, so all PEs will interrupt the current work and set the `SWITCHER_DONE` variable to zero roughly in the same time, which is long before the switcher module sets the variable to one.

One last thing needed is that the activated modules which were previously inactive should load one of the states of the previously active modules. Therefore, the boot procedure should end with the code in Fig. 4.14. The `DESTRESS_MODE` variable should be set to one only once, when entering de-stress mode, and reset to zero when leaving.

State recovery support

OVI_i and EVI_i interrupts, as said, signal voter disagreements in the OML and IML, respectively. They could be triggered only if an NMR group is formed i.e., if the voters are active. In different systems, or, more precisely, different NMR configurations of groups, these interrupts could have different priorities and meanings. The appropriate handlers could be very diverse, from a simple bookkeeping of a single disagreement in an NMR group with greater N (e.g., $N \geq 4$), to a systematic state recovery, reset of modules and restarting operation of the entire NMR group. For example, Fig. 4.15 shows the OVI_1 handler in a FMP(4) system, in which a TMR group is formed, and where the state recovery mechanism of Subsection 3.2.2 is employed. As discussed there, the fourth module in this scheme should be inactive.

Since the system actually sees one module in an NMR group, all modules will have the same `SELF` identifier. Therefore, configuring the FC to interrupt all modules on a single voter disagreement would be the most appropriate choice for this state recovery scheme. As the modules are tightly-synchronized (on a clock-cycle basis), they will

```
void DTIHandler()
{
  store 0, [SWITCHER_DONE];        //set synchronization variable to 0
  saveProcessState(SELF);               //save state of active modules
  int NRACT = nrActive();           //get the number of active modules
  *int OLD_ACTM = getActiveModules();             //get active modules
  *int AGES = getAgeModulesAll();             //get age of all modules
  *int YOUNGEST = youngestFirst(AGES); //age-based sorting of modules
  store NRACT, [NRACT_ADR];                 //maybe needed after reboot
  store OLD_ACTM, [OLD_ACTM_ADR];           //maybe needed after reboot
  store YOUNGEST, [YOUNGEST_ADR];           //maybe needed after reboot
  bool SWITCHER = ifDestressSwitcher(SELF);       //check if switcher:
  if(SWITCHER)                                       //a single module
  {                                           //has to switch all others
    bool SWITCHINACT = 0;                    //inactivate switcher too?
    int i = 1;
    foreach YOUNGEST                          //loop through YOUNGEST:
    {                                  //activate NRACT youngest modules,
      if(i > NRACT)                            //inactivate all the rest
      {
        if(i = SELF) {SWITCHINACT = 1;}
        else {powerOff($_);}
      }
      else {activate($_);}
      i = i+1;
    }
    setDeStressPeriod(DESTRESS_PERIOD);          //activate timer again
    store 1, [SWITCHER_DONE];             //switcher module did its job
    if(SWITCHINACT) {powerOff(SELF);}                 //if not youngest
  }
  else                                  //wait until switcher does its job
  {
    loop while (SWITCHER_DONE = 0);
  }
  for i in 1 to NRACT            //load states of newly active modules
  {                         //with the states of previously active ones
    if(YOUNGEST(i) = SELF) {loadProcessState(OLD_ACTM(i));}
  }
}
```

Figure 4.13: De-stress timer interrupt handler. Assuming YFRR implementation with PG. The SELF identifier is simply the number of the module that executes the procedure.

```
...
if(DESTRESS_MODE)
{
  int NRACT;
  *int OLD_ACTM;
  *int YOUNGEST;
  load NRACT, [NRACT_ADR];                    //stored by DTIHandler
  load OLD_ACTM, [OLD_ACTM_ADR];              //stored by DTIHandler
  load YOUNGEST, [YOUNGEST_ADR];              //stored by DTIHandler
  loop while (SWITCHER_DONE = 0);            //wait here for switcher
  for i in 1 to NRACT          //load states of newly active modules
  {                          //with the states of previously active ones
    if(YOUNGEST(i) = SELF) {loadProcessState(OLD_ACTM(i));}
  }
}
if(FAULT-TOLERANT_MODE)
{
  int NMR_ID = getNMRID();
  loadProcessState(NMR_ID);
}
```

Figure 4.14: End of boot procedure

simultaneously jump to the interrupt handler. The erroneous module is outvoted. After saving their state, the modules firstly investigate which is the erroneous one, by examining the last action register. Then, they activate the inactive module, inactivate the erroneous module, and redefine the TMR group. Synchronization is done by resetting all modules through the FC command register, and loading the saved state. Fig. 4.14 also shows the boot procedure part related to the fault-tolerant mode.

Fault classification support

Two facilities are handy for implementation of the fault classification scheme elaborated in Subsection 3.2.3: the error timers and counters. ETI_i interrupts signal the timer-out events of the error timers which are set to count down the error time period. The error counters automatically register each module error. Of course, this functions only in fault-tolerant mode. Fig. 4.16 shows exactly the same algorithm as in Fig. 3.10.

Note that only one error timer is enough for the fault classification function. Moreover, the de-stress timer could be used in fault-tolerant mode exactly for this purpose – the DTI Handler would be extended with the contents of the ETI Handler. The extended DTI handler would have to additionally check the mode of operation by examining the `DESTRESS_MODE` and `FAULT-TOLERANT_MODE` variables. However, for more advanced FMP systems in which a mix of modes is possible (e.g., de-stress of a NMR group), or, in which several NMR groups could be formed (e.g., two TMR

```
void OVI1Handler()
{
  saveProcessState(SELF);            //save state of the old TMR group
  int INACTIVE = 0;                       //inactive module identifier
  for i in 1 to 4
  {
    if(ifInactive(i)) {INACTIVE = i}       //identify inactive module,
    if(ifModuleError(i)) {clockOff(i)}   //inactivate erroneous module
  }
  *int DMR_GROUP = getActiveModules(); //get the error-free DMR group
  formNMR(DMR_GROUP, INACTIVE);                  //redefine TMR group
  activate(INACTIVE);       //activate module after putting it in TMR
  resetModules(DMR_GROUP, INACTIVE);         //reset the new TMR group
}
```

Figure 4.15: OVI_1 interrupt handler. TMR in FMP(4) with state recovery scheme of Subsection 3.2.2 assumed. CG is used for fast core (de)activation.

groups in FMP(6)), several timers would be needed. Here, it is assumed that P error timers exist, which should simultaneously interrupt the corresponding modules of the NMR group. The handler dispatcher should direct all ETI_i interrupts to the same handler (shown in Fig. 4.16).

At last, the ECI_i interrupts signal an overflow of the error counters. Actually, this interrupt should never be triggered – if it does occur, it is a sign of a serious system flaw.

4.3 Application layer

The application layer should be tailored according to the applications needs, and as such, it is the most diverse since it can use the underlying FM and FC in different ways. In the thesis, three basic modes of operation were assumed – de-stress, fault-tolerant, and high-performance.

4.3.1 Operating modes

The application layer simply sets some variables like `DESTRESS_MODE` and `FAULT-TOLERANT_MODE`, and invokes the FM procedures listed in Table B.1 in order to set the desired mode and set up the system. As said, this should be done according to the application requirements of performance, fault tolerance and power consumption, and the objective of prolonging the lifetime of the system.

For example, let's assume that an FMP(4) system is by default put in high-performance mode after reset. In order to put it into de-stress mode with one active and three inactive (powered-off) modules, three simple operations are enough:

```
void ETIHandler()
{
  bool DO_TESTS = 0;
  bool PERMANENT_FAULT = 0;
  *int NR_ERRORS = getErrorCountAll();       //read all error counters
  foreach NR_ERRORS
  {
    if($_ > ERROR_THRESHOLD) {DO_TESTS = 1}
  }
  if(DO_TESTS) {PERMANENT_FAULT = doExtensiveTests()}
  if(PERMANENT_FAULT)
  {
    ...
    power-off module(s) with permanent fault diagnosis;
    re-form NMR group using healthy modules;
    ...
  }
  resetErrorCountersAll();
  setErrorPeriodAll(ERROR_PERIOD);           //start error timer again
}
```

Figure 4.16: ETI interrupt handler. The `ERROR_THRESHOLD` and `ERROR_PERIOD` are two critical parameters that should be carefully chosen. `doExtensiveTests()` does deep analyses, performs BIST and other self-check procedures that should determine whether a permanent fault occurred in one or more modules. Module(s) with diagnosed permanent faults are powered off. If possible, a new NMR group is formed using only healthy modules. The error counters are reset, and the error timers are started again.

```
powerOff(2, 3, 4);                  //power off all but one module,
setDeStressPeriod(DESTRESS_PERIOD);    //set+start de-stress timer,
store 1, [DESTRESS_MODE];             //store the mode of operation,
```

where `DESTRESS_PERIOD` is a variable (or constant) which is set during the boot-up. After this, the DTI handler (see Fig. 4.13) is the only code needed to implement the YFRR de-stress scheme.

Let's assume that now the system should be reconfigured in fault-tolerant mode with a TMR group. The following commands suffice.

```
noDeStress();                              //turn-off de-stress timer
store 0, [DESTRESS_MODE];                        //no de-stress mode
formNMR(1, 2, 3);                                 //form the TMR
store 1, [FAULT-TOLERANT_MODE];          //it is fault-tolerant mode
activate(2, 3);                     //power-on 2 more modules for TMR
resetModules(1, 2, 3);                    //reset TMR to synchronize
```

Returning to high-performance mode with all modules is simply done by the following commands.

```
noNMR();                                                    //ungroup TMR
store 0, [FAULT-TOLERANT_MODE];                  //no fault-tolerant mode
activate(4);                                     //power-on the 4-th module
resetModules(1, 2, 3, 4);                  //reset all modules through FC
```

Of course, it is assumed that during boot time, the FC is appropriately configured, i.e., action registers, interrupt masks, predefined/inactive outputs, etc., are set.

However, the framework enables far more complex dynamic reconfigurations of the system, starting from a NMROD scheme, to various, mixed modes of operation. For instance, it is possible to have a mix of de-stress and fault-tolerant mode: let in a FMP(9) system only three modules which form a TMR are active, and let the entire TMR is de-stressed. That is, after each `DESTRESS_PERIOD`, a fresh triple of modules replaces the old TMR triple.

Furthermore, the application layer could bring the system to a very high level of sophistication! For example, compared to a common multiprocessor, a fork of a new process in a framed multiprocessor would optionally require stating the required level of fault tolerance which is by default set to a pre-defined value. Then, based on the workload, the framed multiprocessor tries to fulfil the fault-tolerant and performance requirements at the lowest possible rate of aging and power consumption. That is, the application layer tries to find an optimal number of active cores, which dynamically form NMR groups if needed. Additionally, de-stress patterns may be involved. Thus, the framed multiprocessor could execute existing software with no (or minimal) changes.

Nevertheless, the application layer is barely in the scope of this thesis, and is left as a future work (see Section 7.2). The thesis heavily deals with the lower layers though – the framework controller and middleware.

4.3.2 Lifetime-aware task mapping and scheduling

Typical goals of task mapping and scheduling in a common multiprocessor is to reduce the total execution time, increase the throughput, ensure fairness between programs, give priority of execution to specific tasks, etc. Additional goal for the framed multiprocessor would be to reduce aging (improve lifetime). Lifetime-aware task mapping and scheduling is actually a widely investigated topic (see Section 2.2.2).

Generally, the application layer could also employ mechanisms of fault tolerance, lifetime improvement and power saving. For example, lifetime-aware task mapping and scheduling could be a significant supplement of de-stressing, since de-stressing could not be done if all modules are constantly active, e.g., when operating in high-performance mode. A significant advantage of the framed multiprocessor is that it possesses aging monitors which supply information that could be used in the mapping/scheduling process. Nevertheless, this topic is out of the thesis' scope and is left as a future work.

4.4 Design method for programmable NMR voters

This Section details the design method of the programmable NMR voters introduced in Subsection 4.1.4 (see also [SHKK12]), using the same definitions, symbols and denotements. It further gives the performance and area analyses as a function of the redundancy degree N and the input/output width W. Note that software voters implemented using this method are evaluated in [SH13].

The method is based on a binary matrix which represents the equality between the inputs. Actually, the information needed to determine all voter outputs is found in this matrix. Subsection 4.4.1 presents the matrix and its properties, and describes how voter outputs are determined, while Subsections 4.4.2 and 4.4.3 give the performance/area analyses and hardware implementation results, respectively.

4.4.1 Matrix construction and properties

An $N \times N$ binary matrix $\mathbf{A} = [A_{ij}]_{i=0,j=0}^{N-1,N-1}$ is built as follows. If input $x_i = x_j$ then $A_{ij} = A_{ji} = 1$ otherwise $A_{ij} = A_{ji} = 0$, $i, j = \overline{0, N-1}$. For example, let $N = 4$, $x_0 = 20$, $x_1 = 30$, $x_2 = 20$, $x_3 = 10$, and let all inputs be active ($pb_i = 1$). The matrix would be:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

Note that each voter input x_i which is defined as inactive by using the pb_i programming bits, is considered to be different from all other inputs (no matter if active or not).

The following properties of a matrix built in this way are obvious immediately.

Property 1. $\mathbf{A} = \mathbf{A}^T$. $\mathbf{A}$ *is always symmetric, which implies that it is a hermitian matrix* $\mathbf{A} = \mathbf{A}^\dagger$ *and that its eigenvalues are always real numbers* $\lambda_i \in \mathbb{R}$, $i = \overline{0, N-1}$.

Property 2. $A_{ij} = 1$ *for* $i = j$. *All elements of the main diagonal are 1 (all elements are equal to themselves). This implies that* $\texttt{trace}(\mathbf{A}) = N$.

Property 3. *If all inputs are different, then* $\mathbf{A} = I$ *(identity matrix). On the other hand, if all inputs are equal, the matrix elements are all ones* ($\mathbf{A}_{ij} = 1$).

The following assertions which are not so obvious also hold.

Assertion 1. $\mathbf{A}$ *is positive semi-definite matrix.*

Assertion 2. $\lambda_i \in \mathbb{N}_0$. *That is, all eigenvalues of* $\mathbf{A}$ *are natural numbers or zero.*

The proofs of assertions 1 and 2 are beyond the scope and are actually rather long. However, the method is mainly based on the following assertion.

Assertion 3. *The biggest number of equal inputs is equal to the maximal eigenvalue of* $\mathbf{A}$. *(Put in a more comprehensible form: the biggest number of equal inputs is equal to the maximal number of ones in a row (column)).*

Proof. (by Dr. Elena Hadzieva)

Let the most frequent voter input appears r times, that is

$$\begin{aligned} \exists k_1, k_2, \dots k_r \in \{0, 1, 2, \dots, N-1\}, \\ x_{k_1} = x_{k_2} = \dots = x_{k_r} \end{aligned} \tag{4.2}$$

Generality is preserved even in the case when $k_1 < k_2 < \ldots < k_r$.

Since $\mathbf{A}$ is positive semidefinite matrix, all of its eigenvalues λ_i are nonnegative, so its spectral radius $\rho(\mathbf{A}) = \max\limits_{i=0,1,\dots,N-1} |\lambda_i| = \max\limits_{i=0,1,\dots,N-1} \lambda_i$. The inequality

$$\rho(\mathbf{A}) \leq ||\mathbf{A}|| \tag{4.3}$$

holds for every norm of $\mathbf{A}$ (Meyer [77], p. 497). If the $||\cdot||_1$ – norm is selected, then

$$\rho(\mathbf{A}) \leq ||\mathbf{A}||_1 = \max_j \sum_i |A_{ij}| = r, \tag{4.4}$$

since the largest absolute row sum in $\mathbf{A}$ is r.

On the other hand, according to the Collatz-Wielandt formula for nonnegative matrix (Meyer [77], p. 670),

$$\rho(\mathbf{A}) = \max_{\mathbf{y} \geq 0,\, \mathbf{y} \neq 0} f(\mathbf{y}), \quad f(\mathbf{y}) = \min_{0 \leq i \leq N-1,\, y_i \neq 0} \frac{[\mathbf{A}\mathbf{y}]_i}{y_i},$$

where y_i is the i-th component of the N-dimensional vector $\mathbf{y}$ and $[\mathbf{A}\mathbf{y}]_i$ is the i-th component of the N-dimensional vector $\mathbf{A}\mathbf{y}$.

Let the vector $\bar{\mathbf{y}}$ be defined as

$$\bar{y}_i = \begin{cases} 1, & A_{k_1,i} = 1 \\ 0, & A_{k_1,i} = 0 \end{cases}, \qquad i = \overline{0, N-1}$$

which is equivalent to

$$\bar{y}_i = \begin{cases} 1, & i = k_1, k_2, \dots, k_r \\ 0, & \text{otherwise} \end{cases}, \qquad i = \overline{0, N-1}.$$

Then

$$[\mathbf{A}\bar{\mathbf{y}}]_i = \begin{cases} r, & i = k_1, k_2, \dots, k_r \\ 0, & \text{otherwise} \end{cases}, \qquad i = \overline{0, N-1},$$

and

$$\rho(\mathbf{A}) = \max_{\mathbf{y} \geq 0,\, \mathbf{y} \neq 0} f(\mathbf{y}) \geq f(\bar{\mathbf{y}}) = \min \underbrace{\{r, r, \dots, r\}}_{r} = r. \tag{4.5}$$

Now, inequalities (4.4) and (4.5) imply that

$$\rho(\mathbf{A}) = \max_{i=0,1,\dots,N-1} \lambda_i = r, \tag{4.6}$$

which concludes the proof. □

Construction of ISD

Enough information for the state of the voter inputs is given by the elements of the **A** matrix which are above (or below) the main diagonal. This follows from Properties 1 and 2. The elements above the main diagonal of the **A** matrix in Eq. 4.1 are:

$$\begin{matrix} 0 & 1 & 0 \\ & 0 & 0 \\ & & 0 \end{matrix}$$

If the missing places are filled with zeros, a reduced $(N-1)\times(N-1)$ matrix

$$\mathbf{AR} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.7}$$

is obtained, where the original row and column enumeration (of the **A** matrix) for the elements above the main diagonal is kept. That is $i = \overline{0, N-2}$ and $j = \overline{1, N-1}$.

The ISD signals *nr_diff*, e_i and *amb* could be now easily determined. According to Assertion 3, $nr\_diff = \min\limits_{i=0,1,\ldots,N-2} |\{A_{ij} | A_{ij} = 0, j = \overline{1, N-1}\}|$. At the beginning, e_i is initialized to pb_i, i.e., $e_i = pb_i, \quad i = \overline{0, N-1}$. By convention, $e_i = 1$ signals that $x_i \neq y$, while $e_i = 0$ signals that $x_i = y$. Now, all rows $i = \overline{0, N-2}$ of the **AR** matrix are examined, in order to find a row $i = I$, with the smallest number of zeros; here, the zeros which come from inactive inputs are not counted. The output of voting is $y = x_I$, which implies that $e_I = 0$. The rest of the error signals e_j, for $j \neq I$ are determined by examining the columns $j = \overline{1, N-1}$ of row I. That is, $e_j = 0$ only if $A_{Ij} = 1$, otherwise e_j keeps the initialization value. If more than one row with the smallest number of zeros is found, then $amb = 1$, else $amb = 0$.

For example, the row $i = 0$ of the **AR** matrix in Eq. 4.7 has the smallest number of zeros. That means, $y = x_0 = 20$, $nr\_diff = 2$ (there are two zeros in row $i = 0$), and $e_0 = 0$. The rest of the e_j signals for $j \neq 0$ are determined by examining row 0: $A_{01} = 0 \implies e_1 = 1$, $A_{02} = 1 \implies e_2 = 0$, $A_{03} = 0 \implies e_3 = 1$, reflecting which input is equal to the output of voting. Since only row 0 has the smallest number of zeros, the situation is not ambiguous, i.e., $amb = 0$.

Self-check operation

The ISD was easilly determined using Assertion 3, but also using the fact that the **A** matrix (and **AR** in a way) represent a transitive relation:

$$\text{if} \quad x_i = x_j \wedge x_i = x_k \quad \text{then} \quad x_j = x_k. \tag{4.8}$$

Relation 4.8 enables building self-check functions, which is demonstrated by the following example. Let $N = 4$ and $x_0 = x_1 = x_2 \neq x_3$. The corresponding **AR**

matrix would be

$$\mathbf{AR} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Now, assume that for some reason (e.g., a transient fault), AR_{12} is set to 0 instead of 1, which gives

$$\mathbf{AR} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Such a matrix could not be built according to the explained method. Even worse, it is contradictive. It states that $x_0 = x_1$, and $x_0 = x_2$, but in the same time states that $x_1 \neq x_2$. In other words, relation 4.8 is not satisfied. Such a matrix is labelled as *erroneously built*, or, *contradictive matrix*.

Assertion 4. *The corresponding* **A** *matrix of an erroneously built* **AR** *matrix has at least one eigenvalue which is neither a natural number nor zero. That is,* $\exists \lambda_i \notin \mathbb{N}_0$, $i = \overline{0, N-1}$.

A contradictive matrix is an error indicator which the voter could use in order to do a self-check of its own operation. Namely, for each $i = \overline{0, N-2}$ of the **AR** matrix, and each j and k where $j > i$ and $k > j$, the voter checks if relation 4.8 is satisfied. If so, $err = 0$ else $err = 1$. The proof of Assertion 4 is not given here since it is a bit long, and is actually not relevant for the discussion – the voter simply checks relation 4.8, not if the eigenvalues of **A** are naturals or zero.

However, $err = 0$ does not mean that the voter is 100% operating correctly. It does mean that an error is not detected while building the matrix, but parts of the voter which later use matrix information may err and these errors may not be caught.

4.4.2 Performance and area analyses

An evaluation of hardware voters designed using the proposed method follows. Voter performance is evaluated by estimating the propagation delay expressed through the levels of logic needed to build the circuit. Voter area, on the other hand, is evaluated by estimating the number of basic logic gates that constitute the circuit.

Propagation delay

Main building blocks of the programmable NMR voters are comparison circuits and multiplexers. Comparison circuits take approximately $\log_2 W$ levels of logic, where W is the width of the input signals. Multiplexers also take $\log_2 N$ levels of logic to choose among N inputs.

The voter makes $N(N-1)/2$ comparisons between each of the input signals to build the **AR** matrix. However, all comparisons could be done in parallel, so building the matrix takes approximately $\log_2 W$ levels of logic. Next, for each row of the **AR** matrix, the voter has to count the number of zeros. Instead of counting zeros, a

"random" variable (e.g., a 1 represented with N bits) could be shifted once to the left, each time a zero is encountered in a row. In this way, the number of zeros is obtained in a decoded form, which is not important since the numbers are needed only for relative comparisons. Shifting for a constant number of bit positions takes zero levels of logic since only rewiring is needed (or one logic level if buffers are used). For each row, the voter has to determine the number of positions (0 to N-1) for which the variable will be shifted, and select among 2^{N-1} possible input values – that is $\log_2 2^{N-1}$ levels of logic. Repeating this for each row gives $(N-1)\log_2 2^{N-1} = (N-1)^2$ levels. Furthermore, the voter has to check if the number of zeros of the current column is equal to the number of zeros of the previous column (for an ambiguous situation), and then to check if it is bigger. The order of these checks is not important. Checking equality takes $\log_2(N)$, while checking which is bigger takes $2\log_2 N$ levels of logic. That yields $3\log_2 N$ for these operations. At the end, one level of logic is needed to determine e_j for $j \neq I$, in parallel.

These are the main operations which contribute to the levels of logic. The rest of the assignments take either constant or insignificant levels of logic or are parallel to the described operations. For instance, the self-checks account for N comparisons of two bits made in parallel with the operation of counting zeros. Summing up, the levels of logic are $LoL \approx \log_2 W + (N-1)^2 + 3\log_2 N + 1 = (N-1)^2 + \log_2 2WN^3$. The W parameter has a small impact on the propagation delay since it plays a role only during matrix construction. Its contribution is only $\sim \log_2 W$.

One last step remains. A programmable voter has to take into account the pb_i bits which define which input is active. This could be simplified if each input signal is extended one bit. Thus, building the matrix takes $\log_2(W+1)$. Furthermore, the actual number of active inputs can vary from 1 to N. That is choosing between N values, or plus $\log_2 N$ levels of logic. At the end, when counting zeros, the voter has to shift only if the zero comes from active inputs. This is plus $\log_2 N$ logic levels for each row i.e., $(N-1)\log_2 N$. Now, the final approximation for the logic levels is $\approx \log_2(W+1) + (N-1)^2 + 3\log 2N + 1 + (N-1)\log_2 N + \log_2 N$ i.e.,

$$LoL \approx (N-1)^2 + \log_2 2(W+1)N^{N+3}.$$

Number of basic logic gates

As said before, $N(N-1)/2$ comparisons are needed. The number of gates of a comparison circuit is roughly proportional to W, so the number of gates for building the matrix is $\propto NW(N-1)/2$. Next, for each of the N-1 rows, N-1 zero checks of one bit (which is negligible) have to be performed, as well as an N-bits wide comparison of the number of zeros. That is, the gate count is proportional to $N(N-1)(W/2+1)$. In this case too, the area grows more rapidly with the increase of N (proportionally to the square of N). On the other side, assuming that N is constant, the area increases linearly with respect to W. Making similar simplifications for the programming bits as above, the number of gates needed to build the matrix would be $\propto N(W+1)(N-1)/2$. At the end, determining the number of active inputs takes an area proportional to N,

which finally gives an approximation of the total number of gates:

$$Nr.gates \propto N\{(N-1)[(W+1)/2+1]+1\}.$$

4.4.3 Implementation results

This Section presents synthesis results of voters built according to this method. The synthesis is done for the IHP 130 nm technology. The synthesizer is instructed to do no optimizations with respect to the propagation delay and area.

Tables 4.5 and 4.6 present the figures of the levels of logic (LoL), length of the critical path (CPL) in ns, number of gates, area in μm^2 and number of nets, varying N and W, respectively.

Table 4.5: Synthesis results of programmable NMR voters. W is fixed to 16 while N varies.

N	LoL	CPL (ns)	Nr. gates	Area (μm^2)	Nr. nets
2	17	3,78	80	825	114
3	21	6,61	205	2.076	256
4	35	8,87	406	4.092	474
5	44	11,57	646	6.674	732
6	60	15,17	971	9.977	1.074
7	74	18,36	1.306	13.759	1.427
8	90	22,96	1.836	18.660	1.973

Table 4.6: Synthesis results of programmable NMR voters. N is fixed to 4 while W varies.

W	LoL	CPL (ns)	Nr. gates	Area (μm^2)	Nr. nets
1	31	9,61	231	2.134	239
2	32	9,97	271	2.413	283
4	32	9,93	308	2.758	328
8	32	9,78	312	3.024	348
16	35	8,87	406	4.092	474
32	35	9,00	590	6.195	722

The results for the logic levels and gate count from the tables closely match the two formulas obtained by analyses in Subsection 4.4.2. Note that in the case of gate count (or area), an offset and scale constants have to be added in order to match the actual figures, since the formula only gives the proportionality of area with N and W. These constants can be numerically determined from the table data for each N and W.

For the purpose of comparison, Table 4.7 shows the figures of a traditional 3MR voter (varying W), while Table 4.8 shows the same figures of a programmable 3MR voter designed according to the proposed method. The simplest, traditional 3MR

voter has three inputs (each of them W-bits wide), a W-bits wide voting output, and one bit output signalling majority.

Table 4.7: Synthesis results of a simple, traditional 3MR voter.

W	LoL	CPL (ns)	Nr. gates	Area (μm^2)	Nr. nets
1	6	1,77	11	106	14
2	9	1,93	31	252	37
4	9	2,17	54	451	66
8	7	2,34	52	583	76
16	11	2,75	108	1.190	156
32	12	3,08	209	2.343	305

Table 4.8: Synthesis results of a programmable 3MR voter.

W	LoL	CPL (ns)	Nr. gates	Area (μm^2)	Nr. nets
1	17	5,95	91	804	97
2	17	5,96	108	964	117
4	17	6,28	134	1.207	149
8	17	5,43	159	1.565	186
16	20	5,54	236	2.419	287
32	20	5,89	388	4.146	487

Tables 4.7 and 4.8 show that when W is increasing, the differences between the corresponding levels of logic and gate counts (areas) of the voters are decreasing. Thus, for W=1 the programmable NMR voter has 2,83x levels of logic and 8,27x gates compared to the traditional voter, while for W=32 the figures are 1,67x and 1,86x, respectively.

4.5 FMP(4, 4) chip architecture

The 32-bit version of the core presented in Appendix A is used to build an 8-core FMP, organized as FMP(4, 4), i.e, two FWGs with four cores. The block diagram of the chip is shown in Fig. 4.17. The chip is produced and tested in the IHP 130 nm technology.

The framework controllers implement the functions described in Section 4.1. The two memory interfaces are used to connect the FMP to the memory banks. These interfaces contain arbitration logic that grant access to memory to each of the four cores. Each core or NMR group that needs to communicate with memory sends the request (over the FC) to the memory interface. The memory interface resolves which core will be granted access according to the simple Round-Robin algorithm which does not prioritize any of the cores, thus enabling fair access.

The design is leveraged by replication at two levels. Firstly, the core is replicated four times, and connected to the FC and memory controller, thus building one FWG,

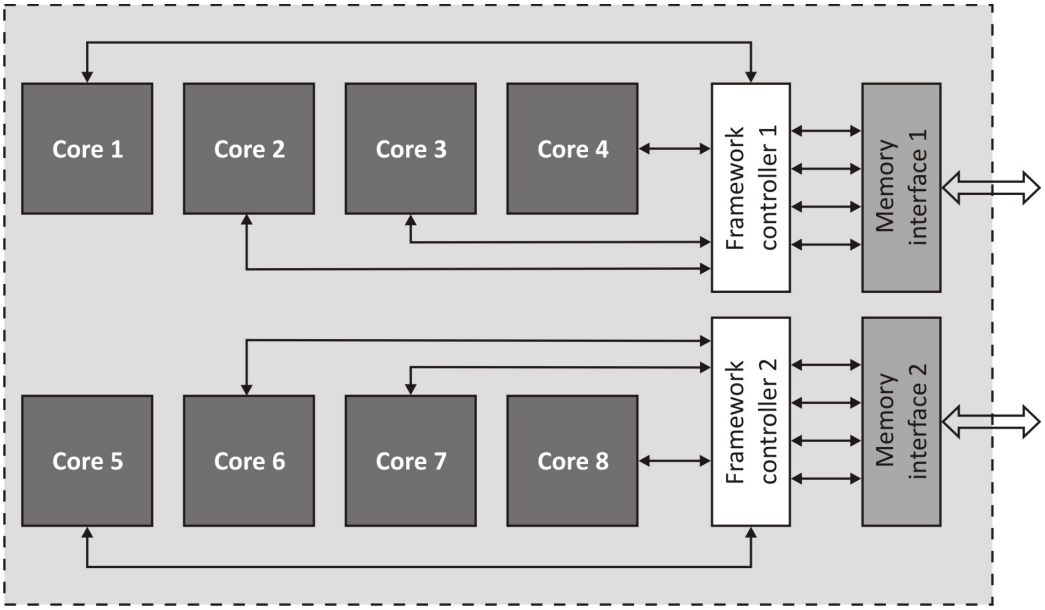


Figure 4.17: FMP(4, 4) chip block diagram. Each FWG has a separate memory interface.

and then, the entire FWG is replicated once again. As can be seen, the design is symmetric with respect to the two FWGs.

Fig. 4.18 shows the layout of the chip. The red areas are caches, while the "cross" section contains the logic. The 208 pads, of which 160 signal pads, direct the layout of the chip, and limit the design in a great extent. The initial target frequency of the FMP was 100 MHz. However, the great power consumption due to the large number of pads led to a decision to reduce the frequency to 50 MHz. Lowering the frequency of operation enabled reducing the pipeline stages from five to two.

4.5.1 Chip performance, power and area

The total area of the chip is 33 mm^2. The area of a single core is $\sim$ 0,5 mm^2 ($\sim$ 1,9 mm^2 with 64 KB L1 cache).

Extensive power reports of the FMP are given in Section 6.3. The FMP has a maximum of 350 mW power consumption of which over 90% is in the pads. That is, $\sim$ 52,5 mW goes into the core of the multiprocessor chip. In order to get an approximate result of a single core power consumption, the last figure should be divided by 9 (8 cores plus framework controllers and memory bus arbitration logic). Thus, a single core with 64 KB cache consumes $\sim$ 5,83 mW. Without cache, the estimate is $\sim$ 1 mW. Power analyses using the PrimeTime tool showed similar results: 7,34 mW and 3,23 mW, respectively.

Performance evaluation is done similarly to the single core in Section A.3. That is, three cases are examined in which all eight processors execute the same copy of a program:

- the caches are not used (switched off);

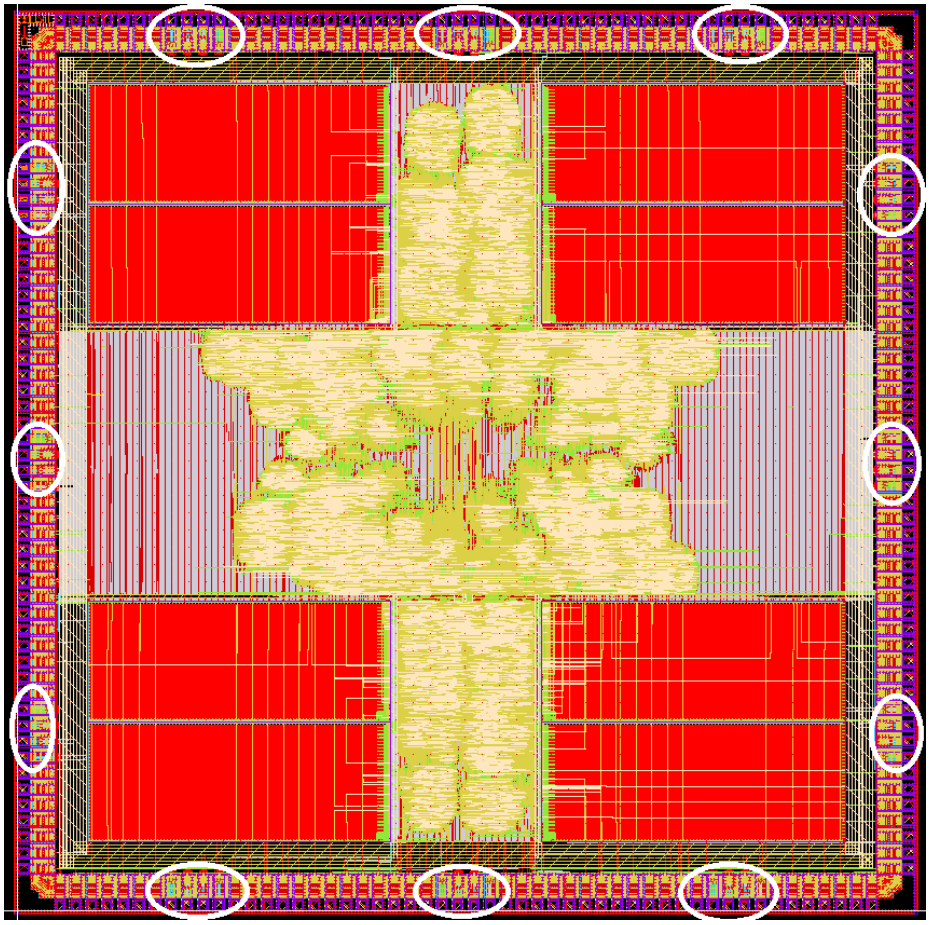


Figure 4.18: FMP(4, 4) layout. The large number of I/O pads constrains the entire design. Power supply pads are encircled.

- the main program loop resides in the L1 caches of each core;
- the main program loop fits completely in one block of L1 (in each core).

In order to get the best cases, the FMP(4, 4) is put in high-performance mode. All eight cores operate in parallel, with no interdependent communication. Table 4.9 shows the average CPI (Clocks Per Instruction) and IPC (Instructions Per Clock).

Table 4.9: FMP(4, 4) performance evaluation

	No cache	Loop in L1	Loop in L1 block
Nr. instructions	1.024.008	1.043.088	1.024.016
Nr. clocks	1.116.352	284.897	212.498
CPI	1,09	0,27	0,21
IPC	0,92	3,66	4,82

Thus, if all eight cores execute their (independent) programs in a block of the L1 cache, a performance of nearly five instructions per clock is achievable.

Chapter 5

Verification environment

A widely used approach [6, 5] for evaluation of fault-tolerant mechanisms and quantification of error resilience is to inject faults in a protected and non-protected circuit and then compare the number of errors that occurred in both circuits under the same conditions. This could offer a fair evaluation of the fault-tolerant mechanism employed in the protected circuit.

Fault injection is a very wide field. It can be performed at many levels during system design. For instance, simulation-based fault injection could take place at the behavioural, register-transfer, gate or geometry level, while emulation-based enables speeding up the fault simulation using FPGAs. Hardware-based fault injection includes methods like heavy ion irradiation, pulsed-laser or electromagnetic interference. A cross-level comparison of fault injection techniques is given in [7].

In order to investigate the fault tolerance against SEE of the proposed multiprocessor framework in this thesis, the following two-step procedure for automated integration of fault injection during the ASIC design flow was used. In the first step, the gate-level netlist of the circuit for fault injection is prepared by inserting saboteurs i.e., Fault Injection Logic (FIL) components at each output of each standard (and non-standard) cell found in the netlist. The FIL components have inputs that control the injection of faults. In the second step, HDL descriptions of fault injectors are generated according to the fault specifications. The fault injectors drive the control inputs of the previously inserted FIL components. In other words, the two steps create a simulation-ready fault injection environment, given the gate-level netlist of the circuit (post-synthesis and/or post-layout) and the fault specifications. Note that the procedures are automated – the single effort required from the user is the fault specification i.e., quoting the fault model, rate and probability of fault occurrence, start/end time of injection, etc.

An appropriate fault injector that drives a FIL component enables modeling of various types of "logic faults" i.e., transient SETs and SEUs (bit-flip, force-0, force-1) and permanent (stuck-at-0, stuck-at-1) faults. It also enables modeling of single event multiple node upsets. Furthermore, if the netlist is not ungrouped (module's boundaries are kept), fault injection could be directed to take place only in specified modules or instances of modules. This gives a possibility to investigate the impact of faults in specific parts on the entire design. For example, faults in a microproces-

sor's data-path have different impact on the microprocessor behaviour compared to faults in its control logic. At last, both stochastic and deterministic patterns of fault injection are possible.

A gate-level simulation is rather slow compared to an RTL simulation, especially for a large circuit. Increasing the complexity of the netlist (by FIL insertion) and by inclusion of fault-injectors further affects simulation time. Nevertheless, the results from the evaluation of this environment show that if the fault rate is not extremely high, the simulation time is 1,17x to 2x greater than the simulation time of the unmodified netlist without fault injection. This price is paid in order to get highly detailed and precise fault injection simulation. It is worth mentioning that the procedure does not change the gate-level netlist of the circuit. It just inserts "vampire taps" that occasionally disturb circuit operation. Thus, simulating the final post-layout netlist could show the real effects of the injected faults in the circuit. The SDF (Standard Delay Format) files that describe the delays of gates and interconnects could be also annotated.

5.1 An overview of fault injection mechanisms

Downside of the simulation-based fault injection techniques is the slow simulation speed if it is not conducted at a high, abstract level as in [65]. On the other hand, emulation-based fault injection in FPGAs such as in [81, 3, 25] enables fast evaluation. However, higher abstract levels do not allow detailed and precise simulation, which is somehow true for the emulation-based approach since the final circuit will be different from the one deployed in the FPGA.

The work in [46] proposes two methods for stochastic fault injection at the gate-level. In the first method, a modified component library that includes fault models is used. Each synthesized gate independently "decides" whether to inject a fault. This approach enables the method to be implemented entirely in the Hardware Description Language (HDL) and facilitates the development of stochastic models for concurrent fault injection. As the authors show, the simulation time is extremely large since each synthesized gate now includes additional code for fault injection. Thus, the method is not suitable for large designs.

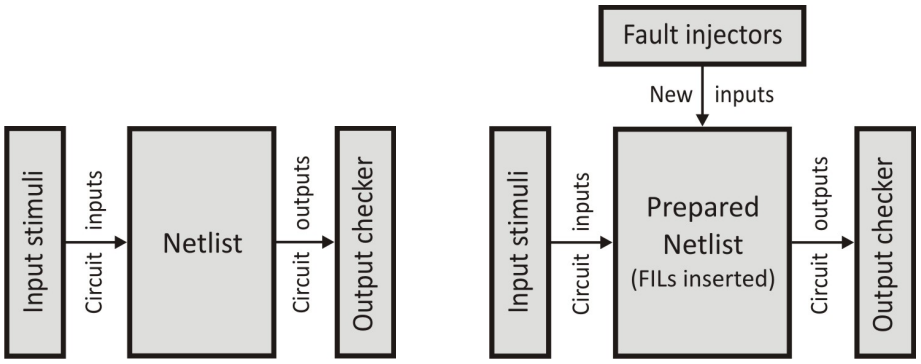
The second proposed method uses a fault injector that randomly chooses gates in the netlist to inject faults. Here, fault injection is centralized, i.e., the fault injector "decides" in which gates to inject faults, enormously improving the simulation speed compared to the previous approach. However, simulation is yet half an order of magnitude slower than simulation of the synthesized netlist without fault injection. The fault injector is written in a high-level programming language and directly interacts with the HDL environment. On the other hand, in the procedure described in this Appendix, the HDL code of the fault injectors is generated automatically, and later integrated into the simulation environment, enabling much faster simulation.

A Verilog-based fault-injection framework is presented in [59]. The authors develop fault libraries in C/C++ that are linked to the simulator using the VPI (Verilog

Programming Interface), which is similar to [46]. Nevertheless, this environment is limited to Verilog and to VPI compliant simulators.

5.2 Automated fault injection procedure

As said, the procedure consists of two automated steps that transform the usual simulation environment from Fig. 5.1(a) into the one shown in Fig. 5.1(b).



(a) Original environment (b) Fault injection environment

Figure 5.1: Preparing a simulation environment for fault injection

5.2.1 Netlist preparation

The gate-level netlist is obtained after synthesis or layout using the target technology library. The technology library has a HDL model (usually Verilog) of the standard and non-standard cells. For simplicity, the cells found in the technology library will be referred to as *leaf gates* from now on.

Fig. 5.2 illustrates how the gate-level netlist is modified for the purposes of fault injection. Each output of each leaf gate in the netlist is rewired to the input of a FIL component (saboteur), and the FIL output is connected to the point of the original gate output. The inputs of the i-th FIL component are labelled with ro_i, the outputs with no_i and the control inputs with $c_{0i}, c_{1i}, \ldots, c_{(N-1)i}$ where N is the number of control bits of the FIL component.

The function of the FIL component is to enable modeling the faults that need to be injected. The example FIL function from Fig. 5.2: $no_i = ro_i\overline{c_{0i}} + c_{1i}(\overline{ro_i} + \overline{c_{0i}})$ allows implementation of both permanent and transient faults. Table 5.1 shows how the logic value of the leaf gate output could be changed i.e., how a fault could be injected using this function. For example, a bit flip or bit inversion is injected if both control signals are held 1 in a given clock period or in a specified time duration. If only one of them is 1, then a logical value 0 or 1 is forced. On the other side, permanent stuck-at-0 faults at a leaf gate output i could be injected if c_{1i} and c_{0i} are being held constantly at 0 and 1, respectively. For stuck-at-1 faults they should

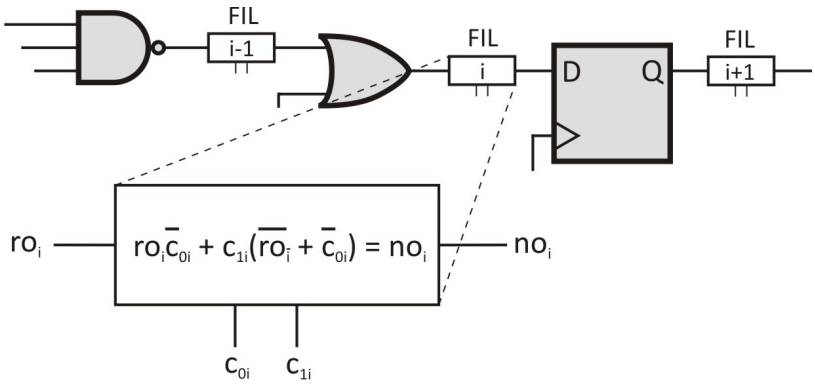


Figure 5.2: Inserting FIL components into a gate-level netlist

be 1 and 0. Furthermore, one could use an arbitrary FIL function with an arbitrary number of control inputs to implement more complex "logical" fault models.

Table 5.1: Truth table of the FIL function in Fig. 5.2

c_{1i}	c_{0i}	no_i
0	0	ro_i
0	1	0
1	0	1
1	1	$\overline{ro_i}$

If LGO is the total number of outputs of all leaf gates in the netlist, then N $\times$ LGO input signals are added to the circuit in order to control the injection of faults (Fig. 5.1). Each of these input signals is connected to one of the control inputs of a FIL component. Of course, for the purposes of simulation, the FIL does not have to be synthesized or implemented with gates of the technological library – the equation suffices.

Now the netlist is ready for fault injection. Next, drivers (fault injectors) for the newly-created inputs of the netlist have to be generated. The fault injectors drive the control inputs of the FIL components, enabling different fault injection patterns and campaigns with various types of fault models.

5.2.2 Generation of fault injectors

According to Table 5.1, one can dynamically choose one or more of the newly-created inputs and inject faults over time by changing the value of the chosen inputs to 1. This could be done in a stochastic or deterministic manner. Changing the value of an input actually changes the value of a leaf gate output. (Of course, if the leaf gate output in a given clock period is 0 for example, and 0 is forced through the chosen control inputs at that leaf gate output in that cycle, the value would not be changed and the injected force-0 fault would not be observed as an error.)

The usual HDL constructs for creating stimuli and testing the design could be used to form fault injectors for each fault group. Since these fault injectors have regular structures, they could be generated automatically by using previously created templates for the fault models. The procedure consists of taking a template copy and filling the numerical values of the fault specification such as rate, probability, start and end of injection.

5.2.3 Fault specification

The term *fault group* to denotes a group of faults with similar characteristics. The user actually specifies fault groups. For each group, the user further specifies the fault type and model, manner, scope and rate of occurrence. The correspondingly generated fault injectors arbitrarily select FIL components i.e., leaf gate outputs and inject the specified faults. Table 5.2 shows the properties of a fault group and some of the possible values.

Table 5.2: Properties of a fault group

Property	**Example values**
Model	bit-flip; force-0; stuck-at-1;
Number of faults	1; 2; 3; ...
Injection frequency	Each 500 cycles; Every 20 μs; Each 100 – 200 cycles;
Probability	1; 0,5; 0,34;
Duration	1 cycles; 2 cycles; 1 ns; 2 ns;
Relative position	pos-0; neg-0; pos-0,12; neg-0,5; pos-1;
Modules affected	all; module_1_name; module_2_name;
Start injection at	50-th clock cycle; 55.000 ns; 0;
End injection at	end; 500-th clock cycle;

The fault models that the FIL function from Fig. 5.2 enables are transient bit-flip, force-0, force-1, permanent stuck-at-0 and stuck-at-1. Both SEUs and SETs could be modelled.

The fault rate could be specified either in clock cycles or time units. It is determined by *number of faults/injection frequency*. E.g., if the number of faults is 3 and the injection frequency is 300 clock cycles, the fault rate is 1/100 faults/cycle. If the injection frequency is 30 μs, the fault rate is 1/10 faults/μs. The number of faults is non-negative integer value.

If the probability property has a value lower than 1 or the injection frequency is defined in the "each-to" manner (e.g., each 100 – 200 cycles, or each 500 – 1000 μs) then the absolute times of the fault injection pulses are specified stochastically. For instance, setting the number of faults to 2, probability to 0,8 and an injection frequency to 300-800, means that there is an 80% probability to inject 2 faults each 300 to 800 cycles.

One could specify the duration of the injected pulses either in number of cycles or in time units. A pure number without time unit denotes number of cycles – this holds for each property that can be specified both in cycles or time units.

The property "relative position" defines the relative time distance from the positive/negative edge of the clock to the starting edges of the injected pulses. E.g., 'pos-0,35' defines that the injected pulse should start after 0,35 * *clock_period* time units after the positive clock edge. The possible values for this property are positive decimal numbers from 0 to 1.

The property "modules affected" defines where faults should be injected – whether in the entire circuit or only in the specified modules or instances. One can specify an array of module and/or instance names in which faults should be injected (an inclusion array), or in which faults should not be injected (an exclusion array). If faults are to be injected in the entire circuit, one could use the special value 'all' for this property.

Start/end injection could be specified either in clock cycles or time units. A special value 'end' denotes the end of simulation and could be used to specify that fault injection should be conducted till the end of simulation.

5.3 Practical implementation

Fig. 5.3 presents the tools and data needed to implement an automated fault injection.

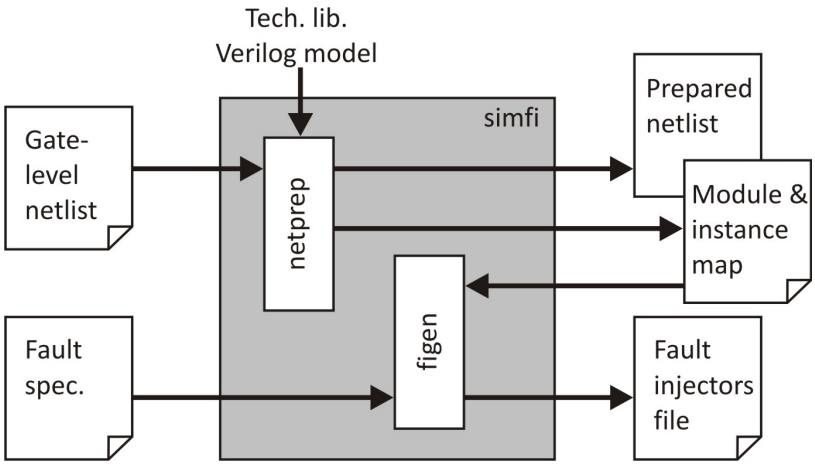


Figure 5.3: Implementation of automated fault injection

Although not shown, the "simfi" script calls both "netprep" and "figen", sets and runs the simulation. Several commercial simulation tools from Cadence, Synopsys and MentorGraphics are used. As can be seen, the single extra effort required from the user is fault specification. That is specifying a few numbers, as explained in Subsection 5.2.3. However, the "simfi" script could be instructed to generate predefined fault groups, so an initial fault injection simulation could be conducted without any efforts.

5.3.1 Netlist preparation – Netprep

At the beginning, the "netprep" script parses the file(s) that contain the Verilog models of the cells in the technology library, archiving the module names and their outputs. Later, this information is used to find the leaf gate outputs in the gate-level netlist and insert the FIL components. Furthermore, the control inputs of each FIL component are added as circuit inputs i.e., the circuit now has additional N input vectors with LGO elements. The LGO parameter is obtained by simply counting the total number of outputs of the used leaf gates in the gate-level netlist. A "dont_insert_fil <leaf gates>" switch may be used to instruct the script to refrain from inserting FIL components at the specified leaf gates. The script outputs the prepared gate-level netlist file.

Additionally, the script outputs the mappings of the control input vector for each module and instance. The "figen" script could later use these mappings to generate fault injectors that target only specific modules/instances for fault injection. This is further explained in Subsection 5.3.2.

5.3.2 Generation of fault injectors – Figen

The "figen" script has predefined templates of HDL code (for now only SystemVerilog) that are used to generate the fault injectors. An example SystemVerilog fault injector for transient bit flips is shown in Fig. 5.4.

```
int r_ind, nr_period = 0, nr_injected = 0;
int lperiod = START_TIME + L_IF_BOUND;
int uperiod = START_TIME + U_IF_BOUND;
int inject_period = $urandom_range(uperiod, lperiod);
forever #CLK_PERIOD begin
  c0 = 0; c1 = 0;
  if(($time >= START_TIME) & ($time <= END_TIME) &
    (nr_period == inject_period)) begin
    repeat(NR_FAULTS) begin
      r_ind = $urandom_range(U_LGO_IND, L_LGO_IND);
      if($urandom_range(100, 1) <= PROBABILITY) begin
        c0[r_ind] = 1; c1[r_ind] = 1; nr_injected++;
      end
    end
    lperiod += U_IF_BOUND; uperiod += U_IF_BOUND;
    inject_period = $urandom_range(uperiod, lperiod);
  end
  nr_period++;
end
```

Figure 5.4: SystemVerilog fault injector for transient bit flips

Input to the "figen" script are the fault specifications. The script generates a fault injector for each specified fault group, setting the corresponding SystemVerilog "localparams" such as `NR_FAULTS` (Fig. 5.4). c0 and c1 are the newly-added control input vectors with LGO elements each. The function `$urandom_range(max, min)` generates random numbers (unsigned integers) in the range between *max* and *min*. Thus, if the fault rate is "2 faults every 1000 to 2000 clock cycles", then `NR_FAULTS = 2`, `L_IF_BOUND = 1000` and `U_IF_BOUND = 2000`. If the fault specification instructs fault injection in specific modules, the script inserts appropriate code and parameters. For instance, if one specifies injection in a module that is mapped in the range from 6500 to 7700, the fault injector always chooses a random index between `L_LGO_IND = 6500` and `U_LGO_IND = 7700` of the control input vectors c0 and c1. If faults are to be injected in the entire circuit, then `L_LGO_IND = 0` and `U_LGO_IND = LGO - 1`. The parameter `PROBABILITY` could be set from 0 to 100 as previously explained. `START_TIME` and `END_TIME` denote the time (clock cycles) when injection starts and ends, respectively.

The script generates the example injector in Fig. 5.4 by a specification of a bit-flip fault model where the numbers for injection frequency, duration, start and end time of simulation resemble clock cycles. If time units were specified, the injector would have different form. Furthermore, different fault models translate to different codes. For instance, if transient force-1 instead of bit-flip is specified, "figen" would not add the "c1 = c0;" line in the code. For force-0 it would use c1 instead of c0. For permanent stuck-at faults the entire structure of the injector would be different. However, all fault injectors that are generated for each corresponding fault group, could be set to drive the newly-added control input vectors c0 and c1. The script actually puts the generated injectors in separate procedures (tasks) that are called in a fork – join block.

5.4 Simulation speed evaluation

Numerous experiments were conducted in order to evaluate this environment. Firstly, the relative simulation time as a function of the fault rate is found by exhaustively simulating a simple ALU. The numbers of injected faults, observed errors as well as some interesting analyses are also given. Then, the relative simulation time as a function of circuit complexity is examined. Furthermore, the possibilities of this platform are demonstrated on the FMP(4, 4) presented in Section 4.5, which is a relatively big design.

5.4.1 Relative simulation time as a function of fault rate

Here, the test vehicle is a simple 8-bit ALU. The number of leaf gate outputs in the ALU is LGO = 2203. The ALU supports the operations of unsigned/signed addition, subtraction, multiplication and division, left/right logic shifts/rotates, right arithmetic shift, and the logical operations: and, or, nand and xor. There are two 8-bit input operands, 5-bit operation selector, 16-bit output, as well as overflow/borrow

and division by zero output bits. The total number of valid input combinations is 796.672. This number multiplied by LGO gives the fault space i.e., the number of possible faults is 1.755.068.416.

The experimental setup is the following. In each experiment (with or without fault injection) an exhaustive simulation is assumed, where all the possible input values are passed to the ALU in separate clock cycles. An output checker counts and reports the errors. The checker reports zero errors when simulating without fault injection. SEU bit-flips are injected with the "probability" parameter set to 100%. The whole netlist is affected and injection occurs during the entire simulation. Table 5.3 shows the simulation results for varying injection rates.

Table 5.3: Results of exhaustive simulations of an 8-bit ALU

Rate (faults/cycle)	Injected faults	Errors	Rel. sim. time
1/600.000	1	0	1,17
1/100.000	7	0	1,17
1/10.000	79	7	1,17
1/1.000	796	52	1,17
1/100	7.966	613	1,18
1/50	15.933	1.266	1,18
1/25	31.866	2.616	1,20
1/10	79.667	6.433	1,22
1/5	159.334	13.246	1,25
1	796.671	65.401	1,58
2	1.593.342	122.182	1,90
3	2.390.013	169.832	2,20
4	3.186.684	212.022	2,49
5	3.983.355	249.413	2,73

The fault injector defines a variable `nr_injected` (Fig. 5.4) that holds the number of injected faults. The output checker reports the encountered errors. According to Table 5.3, the number of errors is an order of magnitude lower than the number of injected faults. This is because the leaf gate outputs are stochastically selected. Thus, a transient fault could be injected in the multiplier during an addition operation. That fault will not appear as an error. The number of errors for the same fault specification is different (but approximate) in successive fault injection campaigns since the function `$urandom_range` arbitrarily chooses times and places of fault injection.

The platform enables various analyses. For example, the number of errors is significantly bigger in the multiplier and divider of the ALU compared to the other circuit blocks because they occupy the most of the space in the ALU (have the greatest number of leaf gates). The significance of such reports is great because they can direct the construction of fault-tolerant mechanisms.

Table 5.3 and Fig. 5.5 show that the simulation time is only 17% greater compared to the simulation time of the original netlist without fault injection for fault rates

below 1/50 faults/cycle. Simulation time rises quickly for extreme rates of fault injection e.g., when injecting several faults in each clock cycle.

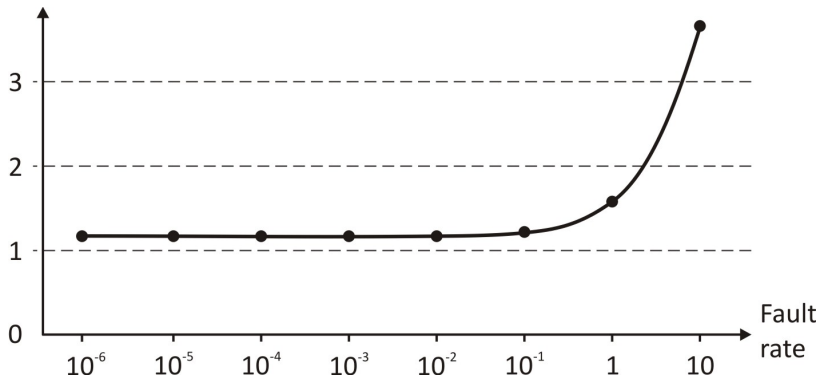


Figure 5.5: Relative simulation times of an 8-bit ALU. Horizontal axis has logarithmic scale with base 10.

5.4.2 Relative simulation time as a function of complexity

In order to represent the relative simulation time as a function of the number of gates in the netlist, the following experiments were set. The ALU from Subsection 5.4.1 is modified by varying the width from 8-bit to 52-bit and/or changing some other parameters and configurations in order to obtain netlists with various number of gates. Many of the ALUs created in this way could not be tested exhaustively in reasonable time. However, functionality is not of concern here, so in each case 3,2M cycles were simulated. Table 5.4 and Fig. 5.6 present the results for varying number of complex (leaf) gates found in the netlists. The number of inverter gates in the corresponding circuits is at least 2,5x bigger.

Table 5.4: Relative simulation times for varying number of complex (leaf) gates and fault injection rates (in faults/cycle)

Nr. gates	**1/1000**	**1/25**	**1/10**	**1/5**	**1**	**5**
20.645	1,43	1,83	2,10	2,63	5,79	6,93
40.180	1,67	2,15	2,67	3,44	6,34	7,35
62.379	1,88	2,25	3,05	3,99	6,78	7,91
81.732	2,04	2,52	3,25	4,45	8,36	9,79
103.297	2,04	2,60	3,46	4,92	9,45	10,80

The rate of 1/1000 faults/cycle is the bottom limit. The same (or very approximate) values are obtained for rates below 1/1000 faults/cycle.

The results from Fig. 5.6 show that for "normal" rates of fault injection i.e., 1/1000 faults/cycle and below, the worst case is maximum 2,04x greater than the simulation without fault injection. Actually, experiments with 1/1000 faults/cycle were made for designs with 200K and 300K complex gates, which gave figures of 1,91x and 1,86x,

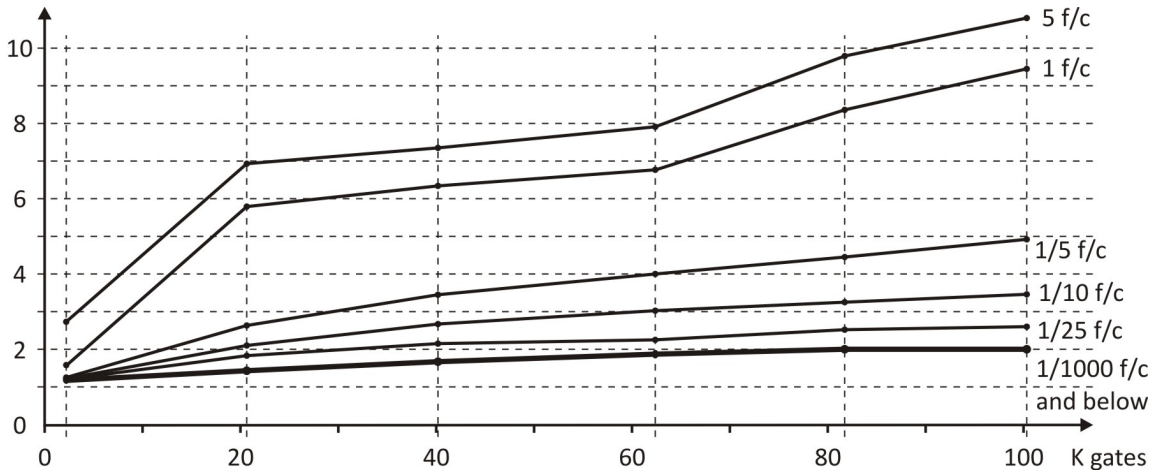


Figure 5.6: Relative simulation time as a function of design complexity

respectively. For small designs, this figure is in the range from 1,17x to 1,25x. Using extremely high rates of injection, e.g., 5 faults/cycle could sky-rocket the simulation time.

5.4.3 Fault injection into FMP(4, 4)

The FMP(4, 4) of Section 4.5 has over 400K complex (leaf) gates (3,1M inverter gates). The simulation setup is similar like in the previous cases. Transient bit-flips were injected in campaigns during which all cores executed the same program. Around 50.000 instructions per core were executed during each campaign. Nevertheless, the exact number of the encountered errors could not be given since the design is too complex to develop exhaustive output and state checkers.

Regarding the simulation time, the following behaviour was observed. If the injection rate is 1/100 faults/cycle or below, the simulation time is 1,44x worse than the simulation time without fault injection (and with the original netlist). This is surprisingly lower value than the one expected from Fig. 5.6. However, the relative simulation time also depends on the testbench complexity. If the fault injectors introduce small activity compared to the rest of the testbench, the relative simulation time goes down. This is the case with the FMP(4, 4) which has complex object-oriented testbench programs and assertion checkers.

If the fault injection rate is higher than 1/100 faults/cycle, figures below 1x are obtained. E.g., for 1/50 faults/cycle the simulation time is 0,80x, which means that simulation is faster with fault injection than without. At first glance, this seems to be a paradox, but what happens is the following. If a fault causes an error that puts a core into an unknown state (where the core signals have unknown X values), the simulator has actually no events to simulate in the X-ed core since the signals remain with the same X values till the end of simulation. Eventually, all eight cores will end up in the unknown state. If the fault injection rates are high, this will happen rather quickly. On the other hand, when the recovery mechanisms successfully mask the

errors i.e., when the fault injection rate is 1/100 faults/cycle or below, the cores do not fall into unknown states and simulation continues with known states and signal values.

5.5 Multiprocessor verification environment

Deep-submicron processes enable vast space for accommodating very complex systems performing complex functions. On the down side, verification of these systems becomes a nightmare. It has been widely reported that verification time accounts from 70% to 80% of the total design time of the chip. Although there are many concepts used to perform verification, increasing complexity means increased number of corner cases which are easy to be forgotten. Nowadays companies have special verification teams, concentrated only to functional verification of the design, apart from the design team. Both teams work in parallel according to the specification of the product which reduces its time-to-market.

5.5.1 (Multi)processor verification techniques

Verification of microprocessors, especially aggressively pipelined super-scalar multi-cores is an extremely challenging task. The reasons as explained lie in the enormous complexity, deeply hiding corner cases.

A detailed description of the verification of the IBM POWER7 microprocessor is given in [103]. The main verification efforts focus on the unit level. Unit level verification is based on Constrained-Random Verification (CRV), using the IBM's cycle-based, event-driven simulator. Additionally, a method for functional coverage and assertion instrumentation, called BugSpray with extended capabilities over the one used for POWER4 is also used to annotate the RTL (Register-Transfer Level) with coverage events.

The verification of another multi-core microprocessor – the SPARC CMT (chip-multi-threaded) with support for 32 threads is presented in [115]. The verification approach and methodology is assertion-based. More than 200.000 assertions (both for coverage and assertion of properties) are developed and maintained during the design cycle, with the "write-once, use-always" philosophy. Simulation-based verification with assertion checkers is also complemented with semi-formal verification to further increase confidence in correctness, using the Magellan formal tool.

The single core, and later the FMP(4, 4) was designed and verified in a special platform [SKK12] based on the *co-verification paradigm*, where software verification starts at the RTL. Thus, hardware is verified along with the software, which significantly reduces time-to-market. Nonetheless, a co-verification environment has even more profound advantages. When a hardware bug or problem is found during software verification, the usual solution is a software patch which almost always degrades performance and/or functionality of the system. If co-verification is used, hardware or software bugs, problems and bottlenecks can be identified early during hardware

design, so they can be resolved before actual chip production, saving efforts, money and time.

Comparison between two co-verification methods of an ARM prototype is presented in [120]. One of them is software-based while the other is based on using hardware-development board. The advantages and disadvantages of speed, degree and insight of simulation are contrasted. Another co-verification platform based on FPGA development board is presented in [125]. The processor is modelled by ISS (Instruction Set Simulator) which communicates with the FPGA through a communication wrapper. A method of co-verification of the 8051 architecture described in SystemC is presented in [104]. The standard comparison of outputs between the HDL (Hardware Description Language) model of the microprocessor and its golden model is used in [22], where large-size real applications are used as test vectors. Finally, there are very powerful commercial environments for accelerating co-verification of ASICs that contain already verified microprocessors, or, interact with them. They are based on hardware emulators assuming that the microprocessor has software ISS (which is slow), encoded RTL model or actual physical component on board (or bonded-out core).

5.5.2 The proposed co-verification platform

Fig. 5.7 depicts the proposed co-verification platform. The software that is intended to be verified could be written either in high-level language or in assembly. Appropriate compiler and/or assembler translates the input code to machine code which is further input to HDL simulator and to the ISS. The ISS is actually a software model of the microprocessor. The output indicates whether errors in one (or both) models are present or not. The platform also enables generating various reports, statistics, data and information about the behaviour of both hardware and software.

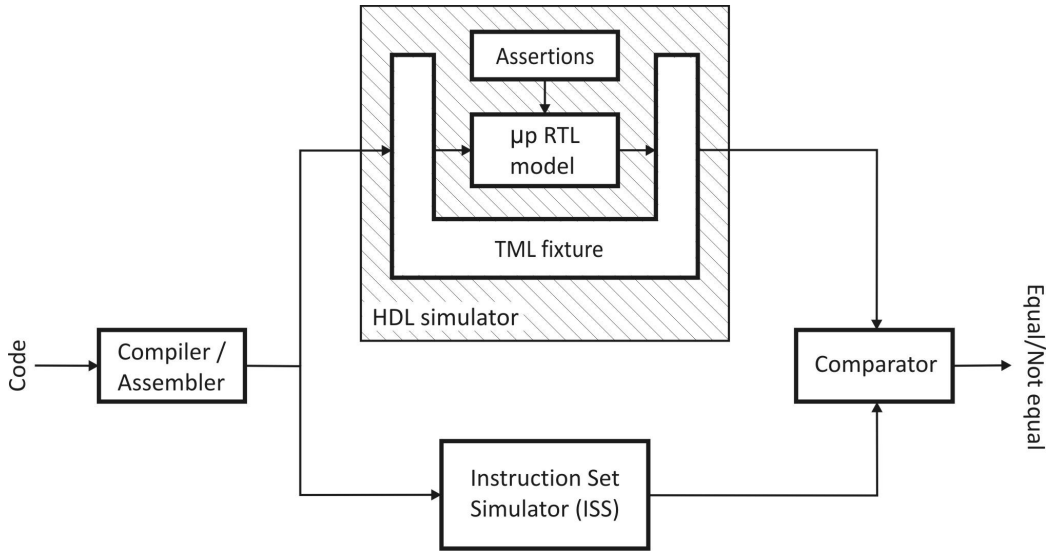


Figure 5.7: HW/SW co-verification, simulation and test platform

The microprocessor's model doesn't need to be even complete in order some portions of software to be executed. While the software is tested and verified, the microprocessor is being verified automatically, finding hardware bugs early in design phase. Also, software problems or bugs can be foreseen. Their solution could be either in hardware or software.

The RTL model of the microprocessor, the models of memories and caches, TML (Test, Monitor and Log) fixture and assertions that verify functionality are written in HDL and simulated by HDL simulator. The TML fixture besides reading the machine code, monitors instruction execution and writes all changes of the microprocessor's state after each executed instruction into a log file. The microprocessor's state is the state of all general purpose, system and control registers, as well as outputs and other internal signals. During code execution, each instruction is checked by at least one assertion which verifies that all changes that are made by the instruction are allowed, and changes that are not allowed are actually not made. For instance, instruction which adds registers reg1 and reg2 and places the result in reg1 should make a change only to reg1, and that change is exactly the sum of reg1 and reg2. All other registers and outputs must not be changed. All instructions are checked by appropriate assertions in this way. Failed assertions have to be examined. Furthermore, functional coverage assertions are used to assess the level of covered cases.

The same machine code translated by the compiler/assembler is input to the ISS which also produces a log of all the changes of the microprocessor's state that each instruction makes. This log should be exactly the same with the one produced by the HDL simulator i.e., the TML fixture. Actually, the same code is input to the same microprocessor models – one written in HDL, and the other in high-level language.

At the back-end, only a comparator of these two logs is needed. A single mismatch in comparison is an indicator that one (or both) of the microprocessor models are erroneous.

Another function of the TML fixture is creation of many other useful reports such as: number of cache misses/hits, number of taken branches, number of correctly predicted branches, average number of clocks per instruction or instructions per clock, number of interrupts and exceptions etc. These reports can be very beneficial both to hardware and software designers since they can point out to hardware/software problems and inefficiencies, trade-offs, performance bottlenecks, not wanted behaviours etc., early in the design phase of the microprocessor. In this phase, it is the cheapest in terms of money, time and effort to agree upon the solutions and make the optimal trade-off.

A feature of this platform is that these procedures are automated. A simple script is used to take the source code of the software as an input, activate the compiler/assembler to translate it to machine code, feed and run the HDL simulator and the ISS, take their output log files and run the comparator to compare them. The result of comparison is displayed, and all reports are already written into files by the HDL simulator and the ISS.

5.5.3 Practical implementation

The platform was implemented and used for co-verification of the 64/32-bit RISC core, and later extended as a multiprocessor platform for co-verification of the FMP(4, 4). The RTL of the designs and the assertions for functional verification are written in SystemVerilog. Cadence's ncverilog HDL simulator is used for simulation and assertion evaluation. The ISS, assembler and comparator are realized in Perl. The script for automation of the processes is also written in Perl. A simple operating system consisting of the basic procedures for context-switching, the framework middleware procedures described in Section 4.2, and other framework-related procedures like recovery mechanisms, is written in the assembly language of the microprocessor (high-level language compiler is under construction). Besides, an instruction generator written in Perl is also used in order to generate large number of valid and random instructions (usually more than 100.000) in order to extensively verify the (multi)processor. The large number of generated instructions is also used to assess and measure the simulation performance of this platform.

Table 5.5 shows results of the simulation times of both ISS and HDL simulator for large number of instructions. ISS and HDL simulator can run in parallel on different machines. Nevertheless, ISS simulation is approximately 10 times faster than HDL simulation, which means that the simulation time is determined by the HDL simulator. Table 5.5 shows that verifying one million instructions takes nearly 12 hours.

Table 5.5: Simulation time in seconds of ISS and HDL simulator. Numbers marked with * are interpolated estimates. Simulation software is running on a four-processor AMD Opteron 856 based server.

Nr. instructions	ISS simulation	HDL simulation
100.000	464	4.150
200.000	907	8.096
300.000	1.313	12.317
500.000	2.278	* 20.750
1.000.000	4.590	* 41.500

Table 5.6 summarizes the time (in person days) that was needed for design and verification of all the hardware and software components.

Table 5.6: Design and verification times of all hardware and software components related to the design, and all components needed for implementing the platform.

COMPONENT	LANGUAGE	PERSON DAYS
Hardware	**SystemVerilog**	**480**
- core RTL	SystemVerilog	270
- TML fixture	SystemVerilog	30
- assertions	SystemVerilog	90
- FMP(4, 4) RTL	SystemVerilog	90
Software	**μp Assembler**	**90**
- operating system	μp Assembler	90
Platform	**Perl**	**104**
- Assembler	Perl	30
- ISS	Perl	60
- Comparator	Perl	2
- Automation script	Perl	2
- Instr. generator	Perl	10
TOTAL		**674**

Chapter 6

Evaluation results

The proposed framework responds to many dependability- and performance-related challenges of multiprocessors. Many aspects have to be investigated in order to fully evaluate the concept, primarily because multiprocessors are very complex systems. These investigations may span across several theses of this size. However, the scope of this work is to at least confirm (or reject) the basic assumptions, and when possible, quantify them. Quantitative evaluation and measurements of the proposed framework regarding lifetime, fault tolerance, power consumption and performance are in the focus of this Chapter. Most of the test cases are based on the FMP(4, 4) implementation presented in Section 4.5.

6.1 Lifetime reliability

Subsection 6.1.1 quantitatively investigates the increase in lifetime of multiprocessors using YFRR, compared to a simple RR, and to no gating at all. Additionally, effects on performance are evaluated in Subsection 6.1.2.

6.1.1 Evaluation of core gating patterns

It is clear that deactivating the cores in a multiprocessor (especially when power gating is used) will prolong system lifetime – the aging effects are negligible when the power supply is off. One of the contributions of the thesis is the following method for analysis of the lifetime increase in multiprocessors that use core gating patterns. The method is based on the Weibul distribution as a function of time t and temperature T_{emp}, as suggested in [1]:

$$R(t, T_{\text{emp}}) = \mathrm{e}^{-\left(\frac{t}{\alpha(T_{\text{emp}})}\right)^{\beta}}, \tag{6.1}$$

where $\alpha(T_{\text{emp}})$ is the scale, and β is the slope parameter. For each core in a homogeneous multiprocessor platform $\beta = 2$. In heterogeneous platforms, $\beta = 2{,}5$ for main processors, while $\beta = 2$ for coprocessors [52].

Let a task be executed by a single core in p time intervals: $\sum_{i=1}^{p} \Delta t_i$. The lifetime reliability of the core at the end of task execution t_p is:

$$R(t_p, T_{\text{emp}}) = \mathrm{e}^{-\left(\sum_{i=1}^{p} \frac{\Delta t_i}{\alpha(T_{\text{emp}})}\right)^{\beta}} \tag{6.2}$$

Now, let the Δt_i intervals be fixed and equal: $\Delta t_i = T, \forall i \in \{1, 2, \ldots, p\}$. Thus,

$$R(t_p, T_{\text{emp}}) = \mathrm{e}^{-\left(\sum_{i=1}^{p} \frac{T}{\alpha(T_{\text{emp}})}\right)^{\beta}} = \mathrm{e}^{-\left(\frac{pT}{\alpha(T_{\text{emp}})}\right)^{\beta}}. \tag{6.3}$$

Eq. 6.3 is in a convenient form for analysis of core gating patterns. T is actually an active period of a core. Let a multiprocessor with N_c cores employ a fair and simple RR pattern. That is, each core is active p/N_c periods. For simplicity, it is assumed $\alpha(T_{\text{emp}}) = \alpha = const$ and $\beta = 2$. The multiprocessor lifetime reliability at the end of mission (at time t_p) is:

$$R_{mp}(t_p) = \underbrace{\mathrm{e}^{-\left(\frac{pT}{\alpha N_c}\right)^2} \mathrm{e}^{-\left(\frac{pT}{\alpha N_c}\right)^2} \cdots \mathrm{e}^{-\left(\frac{pT}{\alpha N_c}\right)^2}}_{N_c} \tag{6.4}$$

or,

$$R_{mp}(t_p) = \mathrm{e}^{-N_c\left(\frac{pT}{\alpha N_c}\right)^2} = \mathrm{e}^{-\frac{1}{N_c}\left(\frac{pT}{\alpha}\right)^2}. \tag{6.5}$$

Let's apply Eq. 6.5 in real scenarios. For example, consider a long-life satellite system with a 10-year mission ($pT = 10$ years). Assume that

$$\left(\frac{pT}{\alpha}\right)^2 = 2. \tag{6.6}$$

Replacing Eq. 6.6 into Eq. 6.5 reveals that a single processor system has a 13,53% probability to survive and fulfil the 10-year mission. Table 6.1 shows the reliabilities of multiprocessors with N_c cores using simple RR gating patterns. Note that in simple RR, the value of T is irrelevant.

Table 6.1: End-of-mission reliability of multiprocessors with varied number of cores N_c using simple RR gating patterns. (Assume Eq. 6.6 holds.)

N_c	2	4	8	16	32	64
$R_{mp}(t_p)$ (%)	36,79	60,65	77,88	88,25	93,94	96,92

So far, the fact that the cores in the system may have different "age" due to phenomena like process variations was neglected. The following definitions are made in order to take this fact into account. Let the age and health state be $a_i(t)$ and $h_i(t)$, respectively, where $a_i(t) = 1/h_i(t)$. These parameters characterize each core $i \in \{1, 2, \ldots, N_c\}$ in the multiprocessor. In the starting moment t_0 of observation of the system $a_i^0 = a_i(t_0)$ and $h_i^0 = h_i(t_0)$.

Alas, it is difficult to obtain absolute values of a_i^0 and h_i^0. Fortunately, their relative values are enough to construct the YFRR pattern, which are easily collected by the aging monitors presented in Subsection 4.1.2. These monitors are convenient for YFRR since they can report the health of each core in the multiprocessor using values which can be represented (translated) in the range from, e.g., 0,5 to 1,5 (±50%),

showing how much the core deviates from the nominal age. Thus, a value lower than 0,5 would mean that the core is dead (out of specs), while 1,5 means that the core is 50% younger than the nominal age of 1. In relative comparison, this would mean that a core with a health state of 0,95 for example, is 5% younger, or, healthier than a core with a health state of 0,9. The relative age range [0,5, 1,5] which is assumed here, could be easily redefined with absolutely no restrictions.

With these definitions at hand, Eq. 6.4 could be extended to include the initial age information of the cores – by simply multiplying the scale parameter α by h_i^0:

$$R_{mp}(t_p) = \prod_{i=1}^{N_c} \mathrm{e}^{-\left(\frac{p_i T}{h_i^0 \alpha}\right)^2} \tag{6.7}$$

Thus, each core reliability is effectively increased/reduced if its initial age is greater/less than 1, leading to greater/lower overall multiprocessor reliability. Note that, generally, the number of active periods p_i may be different for each core as in the case of YFRR, and opposed to a simple RR where $p_i = p, \forall i \in \{1, 2, \ldots, N_c\}$. The same holds for T, but here T is considered equal in all cases.

Now, assume an FMP(4) using a simple RR gating pattern. Let $T = 0{,}1$ years (36,5 days) and let the initial health states h_i^0 are 0,9, 1,25, 0,75 and 1,1. According to Eq. 6.7 the end-of-mission reliability is $R_{mp}(t_p) = 57{,}13\%$. The end-of-mission health states h_i^p are 0,68, 1,03, 0,53 and 0,88, while the single core reliabilities $R_i(t_p)$ are 86%, 92%, 80% and 90%, respectively.

In a real scenario, the aging monitors can always inspect the age of each core, i.e., $h_i(t)$, and based on this information (de)activate the cores according to the YFRR algorithm. However, for the purposes of theoretical evaluation, (in order to calculate p_i) the aging rate $a_r(t)$ should be determined somehow. For convenience, let the aging rate be defined as a percent of aging per active period T. Thus, using Eq. 6.3, the aging rate during period j could be determined as:

$$a_r(jT) = \mathrm{e}^{-\left(\frac{(j-1)T}{\alpha}\right)^\beta} - \mathrm{e}^{-\left(\frac{jT}{\alpha}\right)^\beta}. \tag{6.8}$$

Fig. 6.1 plots Eq. 6.8 using the parameters of the FMP(4). For a better overview, the plot is continuous, although the function is discrete ($j \in \mathbb{N}_0$).

Fig. 6.1 shows that the aging rate reaches a maximum of 1,21% for $j = 50$ (in the middle of the mission). At the beginning, for $j = 1$ the aging rate is minimal – 0,02%. At the end of mission, for $j = 100$ the aging rate is 0,06%.

A small computer program based on Eq. 6.8 is devised to help determining p_i, i.e., determine the active periods of each core in the FMP(4). Inputs of the program are the health states, the mission time and the active period duration T. However, for the purposes of quick paper-and-pen evaluation, assume an average aging rate of $a_r = 0{,}86\%$, derived by averaging Eq. 6.8 for $j \in [1, 100]$. The number of active periods p_i would be 12, 53, 0 and 35 for cores i from 1 to 4, respectively. Replacing this into Eq. 6.7 gives an end-of-mission reliability of 55,01%, which is actually worse than a simple RR pattern. The end-of-mission health states h_i^p are 0,8, 0,79, 0,75 and

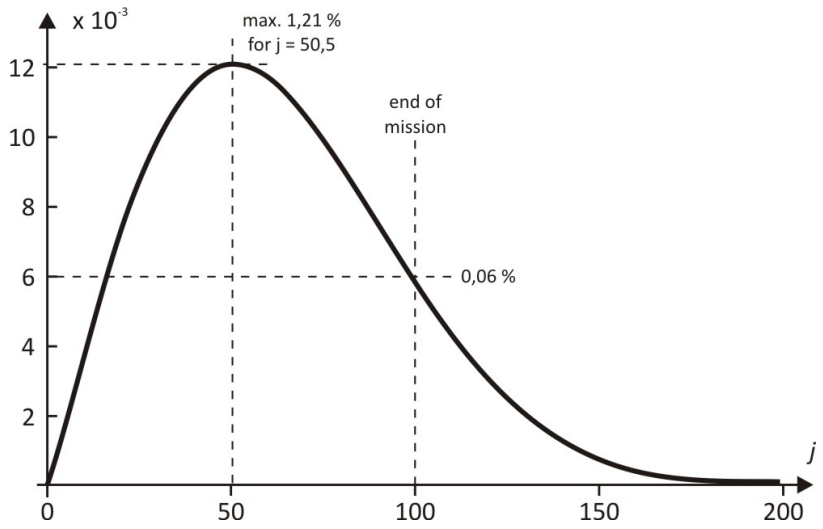


Figure 6.1: Aging rate function of a four-core system with a 10-year mission, with active period $T = 0{,}1$ years. Function scales with T. E.g., if $T = 0{,}01$, y-axis values are 10 times lower, while x-axis values are 10 times greater.

0,8, while the end-of-mission single core reliabilities $R_i(t_p)$ are 96%, 69%, 100% (core 3 is not used) and 81%, respectively.

What happens is the following. Fig. 6.1 shows that the average aging rate is not a linear function of the active period. Too large period may lead to a situation where cores are never activated during the mission (core 3 in the example). Thus, larger initial health state variations require smaller active period T. On the other side, the period can not be too small since frequently switching the power on/off may cause larger stress (and power consumption) than if the core is constantly powered. Performance overheads are also greater. Apparently, the optimal active period has to be found.

In this case, the selected period is too large. Selecting a smaller period of e.g., $T = 0{,}01$ years (87,6 hours) would improve the picture. The numbers of active periods are now 175, 425, 125 and 275, which leads to an improved end-of-mission reliability of 61,42%. Furthermore, more equalized single core end-of-mission reliabilities of 93%, 79%, 95% and 88% are obtained at the expense of less equalized end-of-mission health states of 0,73, 0,83, 0,63 and 0,83. Table 6.2 makes a comparison of the simple RR and YFRR in four cases, varying the initial health states.

As Table 6.2 shows, YFRR excels significantly when large initial variations are present. Furthermore, note that in case IV, simple RR is not able to keep core 1 in the range above 0,5 (which means that core 1 is out of specification). The extreme case where two cores have an initial health state 1,5, while the other two 0,5, shows a 32,92% end-of-mission multiprocessor reliability using simple RR, where two failed cores have an end-of-mission health state 0,25. In contrast, YFRR in the same case achieves 64,12%, without failed cores – two cores have an end-of-mission health state 0,5, while the other two have 1,0.

Table 6.2: Reliability of a four-core multiprocessor using simple YFRR core gating at the end of a 10 year mission. Initial health states h_i^0 are varied. An active period $T = 0{,}01$ years (87,6 hours) is assumed, which means that $p = 250$ for all cores when using simple RR, while average aging rate is $a_r = 0{,}1\%$.

			Simple RR ($p = 250$)			YFRR			
Case	i (core)	h_i^0	h_i^p	$R_i(t_p)$ (%)	$R_{mp}(t_p)$ (%)	p_i	h_i^p	$R_i(t_p)$ (%)	$R_{mp}(t_p)$ (%)
I	1	1,05	0,80	89,28	60,08	267	0,78	87,87	60,81
	2	1,10	0,85	90,19		316	0,78	84,79	
	3	0,90	0,65	85,70		200	0,70	90,60	
	4	0,95	0,70	87,07		217	0,73	90,09	
II	1	0,90	0,65	85,70	57,13	175	0,73	92,72	61,42
	2	1,25	1,00	92,31		425	0,83	79,36	
	3	0,75	0,50	80,07		125	0,63	94,60	
	4	1,10	0,85	90,19		275	0,83	88,25	
III	1	1,10	0,85	90,19	58,29	283	0,82	87,60	61,26
	2	1,20	0,95	91,69		383	0,82	81,57	
	3	0,90	0,65	85,70		184	0,72	91,98	
	4	0,80	0,55	82,26		150	0,65	93,21	
IV	1	0,55	0,3	66,15	47,70	25	0,53	99,59	62,11
	2	0,85	0,6	84,11		125	0,73	95,77	
	3	1,15	0,9	90,98		275	0,88	89,19	
	4	1,45	1,2	94,23		575	0,88	73,01	

Using the same setup as in Table 6.2, Table 6.3 shows extreme cases for varied number of cores N_c, where half of them have initial health states of 1,5, while the other half is set to 0,5.

Table 6.3: Comparing simple RR and YFRR in extreme cases. In each row, $N_c/2$ cores have initial health states of 1,5, while the other half 0,5. The "almost dead" half of cores is never used if YFRR is applied, which means that their end-of-mission health state is also 0,5, opposite to simple RR where the h_{rr}^p column shows the lowest values.

N_c	h_{rr}^p	$R_{rr}(tp)$ (%)	$R_{yfrr}(tp)$ (%)	Δ (%)
2	0,00	10,84	41,11	30,27
4	0,25	32,92	64,12	31,20
8	0,38	57,38	80,07	22,70
16	0,44	75,75	89,48	13,74
32	0,47	87,03	94,60	7,56
64	0,48	93,29	97,26	3,97
128	0,49	96,59	98,62	2,03

Note that the evaluation method presented here was exercised using one assumption which may not correspond to real-life systems i.e., the assumption that Eq. 6.6 holds. This was made with the single purpose to ease the computations. In other words, it was initially assumed that a single processor system has a 13,53% end-of-

mission reliability. However, applying any number in the range of (0, 100)% for the end-of-mission reliability would give proportional results and confirm the superiority of YFRR compared to simple RR. As said, the same holds for the relative age which was defined in the range of [0,5, 1,5].

It is worth mentioning that an interesting analytical result could be obtained for the optimal active period T, which was found here by a computer program. That is, it is possible to find a closed-form expression that gives the optimal period T for which the average aging is minimal. The procedure consists of finding the stationary points, i.e., the roots of the first partial derivative of Eq. 6.8 with respect to T. The function may have maxima, minima or no extrema in the stationary points which is revealed by the second (or higher order) partial derivative with respect to T. Unfortunately, it is very complicated to find the first derivative of this function, which discourages exercising the procedure. Using computer programs like Matlab or Mathematica would be beneficial in this direction.

6.1.2 Effects of core gating on performance

It is clear that the performance overhead due to core gating is directly related to the active period T. That is, the more frequently are the cores gated, the greater are the performance overheads. At the end of each T period, the active cores need to write their states in memory, which will be then read by the newly activated cores (see the DTI handler procedure in Fig. 4.13). Other factors like cache warm-up also should be included in calculating the overheads. However, the evaluation presented in Subsection 6.1.1 shows that the active periods should be very large (days or months), which leads to extremely small performance overheads that are not worth examining. This fact also answers the question whether PG or CG should be used (see discussion in Subsection 3.1.2), i.e., an YFRR pattern should always opt PG.

6.2 Error resilience in fault-tolerant mode

As discussed several times before, the greatest advantage of NMR systems is that they can instantly mask faults and continue operation without invoking any error recovery mechanisms, and incurring no performance penalties. However, a fault that flips the state of a memory element, e.g., a bit in the register file, brings the core into a non-consistent, erroneous state. The voter may see intermittent or continuous disagreements at the outputs of that core. Therefore, recovery mechanisms are still needed. The programmable NMR voter, as described, easily identifies these situations, and may signal the framework controller to initiate an appropriate action for state recovery.

This Section investigates the fault tolerance of core-level NMR systems, both with and without recovery mechanisms. Firstly, Subsection 6.2.2 investigates the time period that the NMR group would function correctly without invoking any recovery mechanisms, under various fault rates. This can be alternatively seen as finding the minimal number of redundant modules in a NMR system needed to survive

the mission time, given the fault rate. Secondly, Subsection 6.2.3 examines several recovery mechanisms, including the one proposed in Subsection 3.2.2.

For these purposes, the following experimental setup is employed.

6.2.1 Experimental setup

The simple RISC core presented in Section A.1 that includes separate 64 KB instruction and data caches is used to build a 16-core framed multiprocessor – FMP(16). The programmable NMR voter dynamically forms 1MR, 2MR,..., 16MR groups. A simple program (see Fig. 6.2) whose length of execution could be easily changed is used as a workload. The program reads two arrays of integers from memory, performs arithmetic operations on each element, and writes the results back to memory. The size of the arrays is the actual parameter that defines the length of execution of the program.

```
void NMRftTestProgram()
{
  *int a, b, c;                                    //arrays of integers
  int NR_ELEMENTS = 10000;                  //nr.  elements in a, b and c
  for i in 0 to NR_ELEMENTS - 1
  {
    c[i] = a[i] + b[i];
  }
}
```

Figure 6.2: Simple test program. Length of execution is easily controlled by the `NR_ELEMENTS` parameter.

The platform for automated integration of fault injection into the ASIC design flow, fully described in Section 5.2, is used in the experiments to inject faults in the synthesized netlist of the FMP(16). This platform enables fine tuning of many parameters like fault rate, probability, periods of injection, etc., and allows defining which of the modules should be a target of fault injection.

A special environment (see Section 5.5 and [SKK12]) is used to keep track of the multiprocessor state. That is, the states of the flip-flops and outputs of each core are logged in each clock cycle. Comparing the log of execution in the case with no injected faults to the logs of executions with faults injected at various rates, shows which of the cores "survived" and whether the NMR system operated with a majority of error-free cores.

6.2.2 Error resilience without recovery mechanisms

Numerous experiments were conducted using the setup described in Subsection 6.2.1 in order to examine the error resilience of the framed multiprocessor operating in fault-tolerant mode. The number of active cores in the NMR system N was varied from 1 to 16. The fault injection rate was varied and correlated to the length of

execution (mission time). Note that faults were not injected in the programmable NMR voter because the only way to recover from voter errors is to trigger the fault recovery mechanisms, which is not wanted here. Both transient and permanent faults were injected.

Injection of transient faults

Tables 6.4 and 6.5 show results for an array of 100 and 1.000 elements, respectively. Similar results were obtained for an array length of 10.000 (execution time of 310.164 cycles), and the corresponding fault injection rates of 1/10.000, 1/5.000, 1/2.500, 1/1.000, 1/500 and 1/250 faults/cycle. In this case, fault rates greater than 1/100 faults/cycle failed all cores for each N.

Table 6.4: Injection of bit-flips. The table shows the number of cores that remained error-free after fault injection with various rates. Grayed cells stress cases where majority is sustained. An array length of 100 is assumed (execution time is $\sim$ 3.100 cycles).

	Fault rate (faults/cycle)				
N	1/100	1/50	1/25	1/10	1/5
1	1	0	0	0	0
2	2	0	1	0	0
3	1	3	0	0	0
4	4	3	2	2	0
5	4	4	2	0	0
6	5	4	4	0	0
7	5	5	3	2	0
8	7	8	8	1	0
9	8	6	3	3	2
10	8	7	7	6	2
11	10	10	8	4	1
12	12	12	10	8	3
13	12	11	9	7	3
14	14	12	11	8	2
15	15	13	13	8	0
16	14	12	11	7	5

Assuming the same (constant) fault injection rate, two obvious comments are the following. Firstly, the probability that the system fails is proportional to the execution time. Secondly, increasing N actually increases the area on which faults are injected. Thus, the same number of faults spread over greater area leads to smaller number of failed cores.

Using the results from these experiments one can easily construct the majority lines for the core-level NMR groups by interpolation. These lines show the fault injection rates and mission times for which the system is able to keep a majority of error-free cores. The graphs are presented in Fig. 6.3. If the $(t, F_r(t))$-point of a given

Table 6.5: Injection of bit-flips. The table shows the number of cores that remained error-free after fault injection with various rates. Grayed cells stress cases where majority is sustained. An array length of 1.000 is assumed (execution time is $\sim$ 31.000 cycles).

	Fault rate (faults/cycle)							
N	1/1.000	1/500	1/250	1/100	1/50	1/25	1/10	1/5
1	0	0	0	0	0	0	0	0
2	2	1	1	0	0	0	0	0
3	2	3	0	0	0	0	0	0
4	4	3	2	2	0	0	0	0
5	4	4	0	0	0	0	0	0
6	6	4	4	0	0	0	0	0
7	5	5	2	1	0	0	0	0
8	8	7	7	0	0	0	0	0
9	8	7	4	3	2	0	0	0
10	8	8	7	6	3	0	0	0
11	10	10	9	3	0	0	0	0
12	10	11	8	9	4	2	0	0
13	12	11	11	5	3	0	0	0
14	13	11	13	8	0	1	0	0
15	15	15	0	6	4	1	0	0
16	14	13	11	8	5	0	0	0

NMR group is below the N-th line, the system will complete execution successfully (sustain majority during the mission). Otherwise, the system will fail (will not sustain majority).

A straight line in a log-log plot is easily converted into a closed-form expression using the template $y = ax^b$, where a is the slope of the line and $\log b$ is the interception on the $(\log y)$-axis. That is, the lines for $N \in [1, 2]$, $N \in [3, 9]$ and $N \in [10, 16]$ in Fig. 6.3 are $y = 1/x$, $y = 2/x$ and $y = 4/x$, respectively. Hence, the closed-form expression of the majority lines as a function of time t would be:

$$F_r(t) = \begin{cases} \dfrac{31}{ft}, & \text{for } N \in [1, 2] \\ \dfrac{62}{ft}, & \text{for } N \in [3, 9] \\ \dfrac{124}{ft}, & \text{for } N \in [10, 16] \end{cases}, \tag{6.9}$$

where $F_r(t)$ is the fault rate, and f is the operating frequency.

However, of practical interest is to determine whether a given $(t, F_r(t))$-point is below or above the majority line, i.e., whether the NMR group will survive or not. For a straight line $Ax + By + C = 0$, a point (p, q) is above the line if $q > -\frac{B}{A}p - \frac{C}{A}$.

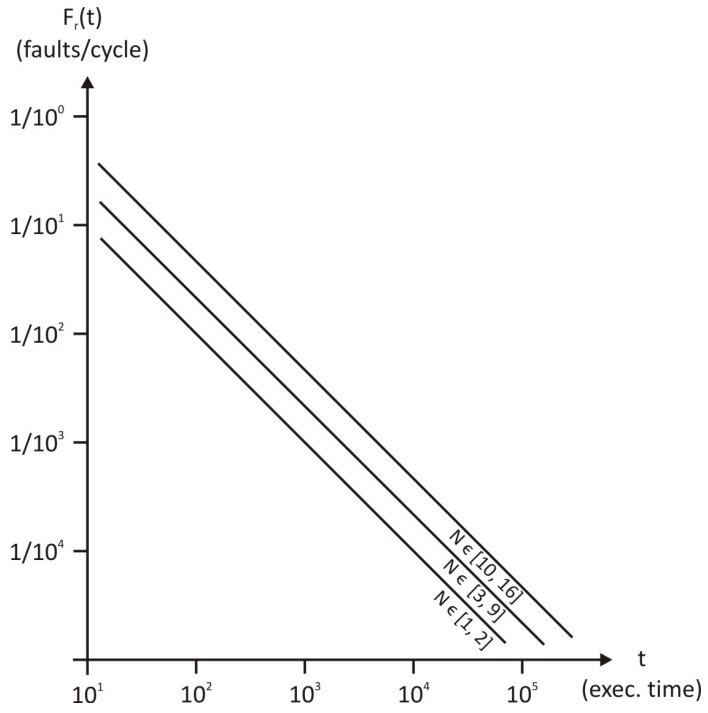


Figure 6.3: Majority lines of core-level NMR groups. Both x and y axes are in logarithmic scale (log-log plot). Note that for simplicity of computation and representation, the x-axis values correspond to the array length. These values should be multiplied by ~ 31 to get the actual number of cycles, and by 1/50 MHz to get the actual execution time in ns (simulation was done at 50 MHz).

Since Fig. 6.3 is a log-log plot, one should investigate the $(\log p, \log q)$ point instead, i.e.,

$$\begin{aligned} &\log q > -10^5 \log p + 1100, \quad \text{for } N \in [1,2], \\ &\log q > -10^5 \log p + 2100, \quad \text{for } N \in [3,9], \\ &\log q > -10^5 \log p + 4100, \quad \text{for } N \in [10,16]. \end{aligned} \tag{6.10}$$

Thus, if the system (or user) knows the length of execution of the timing-critical task, it can assign the appropriate number of cores by exercising Eq. 6.10, and guarantee fault-tolerant operation without invoking recovery mechanisms.

It is worth noting that enabling or disabling the instruction/data caches of the cores does not play a role. Very similar results were obtained in both cases (with and without using caches).

Injection of permanent faults

Exactly the same experiments were conducted with permanent faults. In the case of permanent 'stuck-at' faults, the obtained results are absolutely identical for each investigated array length (100, 1.000 and 10.000), and for the corresponding fault rates. Table 6.6 shows the results.

Table 6.6: Injection of stuck-at faults. The table shows the number of cores that remained error-free after fault injection of stuck-at-0 with various rates. (Stuck-at-1, and mixed stuck-at-0 and stuck-at-1 fault injection simulations show similar results.) Grayed cells stress cases where majority is sustained. The corresponding fault rates in the case of an array length of 10.000 are given, where the execution time is $\sim$ 310.000 cycles.

	Fault rate (faults/cycle)					
N	1/10.000	1/5.000	1/2.500	1/1.000	1/500	1/250
1	0	0	0	0	0	0
2	2	1	0	0	0	0
3	2	0	0	0	0	0
4	2	2	2	0	0	0
5	3	2	0	0	0	0
6	6	4	2	0	0	0
7	6	4	0	0	0	0
8	8	6	4	0	0	0
9	8	6	5	0	0	0
10	8	8	0	0	0	0
11	8	6	0	0	0	0
12	12	11	0	0	0	0
13	12	11	9	0	0	0
14	13	10	9	6	2	0
15	14	13	11	0	0	0
16	15	15	11	0	0	0

As expected, it is harder to cope with permanent faults. That is, greater number of cores are needed to sustain majority, assuming the same fault rate. E.g., for a fault rate of 1/5.000 faults/cycle and an array length of 10.000, transient bit-flips could be masked with three cores, while stuck-at faults with six cores at least.

The same analysis as in the case with bit-flips could be made for stuck-at faults too. A similar equation to Eq. 6.10 is obtained. Here, the lines of plot 6.3 would be shifted towards the (0, 0) point.

6.2.3 Employing recovery mechanisms

An NMR group of cores could achieve much greater reliability regarding transient faults if recovery mechanisms supplement the NMR mechanism itself. Here, three recovery mechanisms that treat transients were examined:

1. Save-reload: save and reload the state of the cores;

2. Save-reset-reload: save the state, reset the NMR group, and then reload the state;

3. Save-reset-reform-NMR-reload: the same as mechanism 2, but at least one additional powered-off core in the FWG is activated to replace the faulty core and reform the NMR group upon a detected error. (See Subsection 3.2.2.)

Actually, all of the mechanisms are based on saving the current state of the core(s) upon a detected error. The state is defined as the contents of all internal registers of the core(s). Of course, only registers which are directly addressable, and both readable and writeable, are taken into account.

Furthermore, all of the recovery mechanisms are triggered by the ISD of the programmable NMR voter (Fig. 4.5), which signals detected errors in one or more cores of the NMR group, but also by the voter's *err* signal which reports the outcome of the self-check operation.

Having an NMR majority, the correct cores in the NMR group will outvote the faulty ones and write the correct state into the memory (assuming no faults occurred out of the NMR group). The last part of all examined recovery mechanisms is to reload the state back to the cores. The procedures for saving and reloading the state are similar to a context (process) switch, with the difference that after recovery, the same process is loaded.

Assuming that the NMR group always sustains majority in presence of errors, the simplest recovery mechanism (1) is to save the state, during which process the erroneous state is outvoted, and then simply reload the correct state to the NMR group. Unfortunately, this mechanism is rarely successful (as the simulations showed). The reason is that there are a lot of other memory elements such as registers or flip-flops, which could not be saved and reloaded in the way the state registers are. Thus, the erroneous state of these elements could not be fixed, and the errors quickly accumulate.

Nevertheless, including a reset step between the save and load procedures (mechanism 2) turned out to be sufficient. The simulations showed perfect recoveries for fault rates up to 1/100, and in many cases up to 1/50 faults/cycle. Larger fault rates affect the recovery procedure itself since the duration of saving and reloading the state is around 200 cycles.

Resetting the cores is practically performed by invoking the `resetModules` function of the framework middleware after saving the state. The function sets the corresponding bits in the framework controller's command register that further triggers the reset lines of the cores in the FWG (see Table B.1 and Fig. 4.9). After reset, the boot procedure checks to see who made the reset by investigating the FC registers, and then reloads the saved state of the cores. Of course, the boot procedure knows the predefined memory location of the saved state.

Finally, mechanism 3 achieves the same results as mechanism 2, but its advantage is that it de-stresses the core(s) in which errors were detected. For example, if NBTI effects were responsible for the fault, intermittent errors may be avoided since the faulty core would be left inactive for a while, recovering from NBTI.

On the downside, recovery mechanisms introduce performance penalties which are examined next. Tables 6.7 and 6.8 show the overheads of mechanism 2 for various fault rates, with and without using the L1 caches, respectively. Mechanism 3 exhibits

almost completely equal overheads, which is expected since its procedures have just several instructions more than mechanism 2. Note that the number of active cores in the NMR group is irrelevant regarding the performance, as long as there is majority. Therefore, all experiments were made with $N = 16$ using the setup described in Subsection 6.2.1. The array length is 10.000 in all cases.

Table 6.7: Performance overheads of state recovery mechanisms. Caches are inactive. (Note that in the case without fault injection and recoveries, 110.011 instructions are executed in 310.156 clock cycles.)

				Overhead in %	
Fault rate	Nr. recoveries	Nr. instr.	Nr. cycles	Instructions	Exec. time
1/10.000	1	110.046	310.262	0,03	0,03
1/5.000	2	110.081	310.368	0,06	0,07
1/2.500	3	110.116	310.474	0,10	0,10
1/1.000	13	110.466	311.534	0,41	0,44
1/500	23	110.816	312.594	0,73	0,79
1/250	48	111.691	315.244	1,53	1,64
1/100	121	114.246	322.982	3,85	4,14

Table 6.8: Performance overheads of state recovery mechanisms. L1 caches are active. (Note that in the case without fault injection and recoveries, 110.013 instructions are executed in 318.406 clock cycles.)

				Overhead in %	
Fault rate	Nr. recoveries	Nr. instr.	Nr. cycles	Instructions	Exec. time
1/10.000	1	110.048	319.218	0,03	0,26
1/5.000	2	110.083	320.030	0,06	0,51
1/2.500	3	110.118	320.842	0,10	0,77
1/1.000	13	110.468	328.962	0,41	3,32
1/500	23	110.818	337.082	0,73	5,87
1/250	48	111.693	357.382	1,53	12,24
1/100	121	114.248	416.658	3,85	30,86

When the cores use their L1 caches (separate data and instruction cache per core), execution time overheads are greater since the caches warm up on each reset. Of course, instruction overheads are the same in both cases since the same program is executed.

Note that in the case without fault injection and recoveries, program execution lasts longer in the case when caches are enabled (318.406 vs. 310.156 cycles). This is because the data cache is write-through with no write allocation, which means that each write and then read to/from the same memory location will cause a cache miss. On the other side, the program intensively reads and writes data from/to memory. Using a write-back cache improves this picture, although execution time overheads

when fault injection triggers recovery mechanisms remain the same as write-through – execution time overheads are caused mainly by the instruction cache which has to be reloaded on each reset.

6.3 Power consumption

Using PG and CG will definitively lower power consumption, which is one of the goals of the thesis proposal. This Section presents the results of the quantitative evaluation of power consumption of the proposed multiprocessor framework. The 8-core FMP(4, 4) chip presented in Section 4.5 is used for this purpose. The chip was successfully produced and tested in the IHP 130 nm technology with the target frequency of 50 MHz. The chip core and pads supply voltages are 1,2 V and 3,3 V, respectively. Note that the chip core contains all of the eight logical processor cores (PEs), as well as the framework controllers and memory interfaces (see Fig. 4.17).

6.3.1 Simulated power analyses

Unfortunately, power consumption measurements of PG could not be performed, since power gating is not integrated into the IHP 130 nm design flow. Thus, the cores in the produced chip could not be power-gated. Nonetheless, power analyses were made using the Synopsys' PrimeTime power analyser. The simulated results of power consumption where the number of powered off cores is varied from 0 to 7 are given in Table 6.9, while the results of clock-gated cores are given in Table 6.10.

Table 6.9: Simulated power analysis of power-gated cores. N is the number of active (power-on) cores.

	Simulated chip core power in mW	
N	Without caches	With L1 caches
1	1,92	2,16
2	2,39	4,32
3	3,58	6,48
4	4,77	8,63
5	6,00	10,85
6	7,17	12,91
7	8,37	15,12
8	9,58	17,38

6.3.2 Chip measurements

Table 6.11 shows the results of on-wafer measurements of power consumption using core gating, where the number of clocked off cores is varied from 0 to 7. Again, two cases with and without active caches are examined.

Table 6.10: Simulated power analysis of clock-gated cores. N is the number of active (clock-on) cores.

	Simulated chip core power in mW	
N	Without caches	With L1 caches
1	2,65	3,07
2	3,13	5,34
3	4,19	7,67
4	5,65	9,83
5	6,78	11,55
6	8,12	14,05
7	9,74	16,29
8	10,31	18,46

Table 6.11: On-wafer power measurements of clock-gated cores. N is the number of active (clock-on) cores.

	Measured chip core power in mW	
N	Without caches	With L1 caches
1	1,56	2,52
2	2,28	3,72
3	3,00	4,92
4	3,60	6,12
5	4,44	7,56
6	5,04	8,76
7	5,88	10,08
8	6,48	11,28

For the purposes of comparing the measured vs. simulated results of power, Table 6.10 gives the simulated results of power consumption using CG, which could be compared to the measured results of Table 6.11. Table 6.12 shows the percentage of deviation of simulated (Table 6.10) vs. measured (Table 6.11) results of CG power consumption. The differences are in the range from 18% to 39%.

6.3.3 Discussion

Chip core power is reduced proportionally when PG/CG is used. Each activated core contributes with approx. 1 mW. PG saves around 0,8 mW more compared to CG. Activating the caches leads to higher power consumption: for PG, the overhead is in the range from 12,5% for one active core to 81,4% for eight active cores, while for CG is from 61,5% to 74,1% (using results from measurements).

One peculiarity of the produced FMP(4, 4) is that over 90% of the power is consumed by the large number of pads, which were chosen to enable strong current drive capability. Pads could not be power- or clock-gated – they should be active even

Table 6.12: Percentage of deviation of simulated vs. measured results

	%Δ	
N	Without caches	With L1 caches
1	41,13	17,92
2	27,16	30,34
3	28,40	35,85
4	36,28	37,74
5	34,51	34,55
6	37,93	37,65
7	39,63	38,12
8	37,15	38,89

if only one core is operating. Thus, the difference between the least power consuming case (one core with no caches), to the most power consuming case (eight operating cores with caches) is only 34,5 mW for CG, and 76,3 mW for PG.

An interesting observation caused by this peculiarity is the following. An expected behaviour is that when caches are used, core power would be increased, but pad power would be decreased, leading to a lower figure of total power consumption. When the caches are used, the frequency of memory requests that go out of the chip is reduced, which lowers the dynamical power consumption by the pads. The conclusion in this case would be that caches should be used in order to reduce power consumption.[1] This behaviour was actually caught by the chip measurements. Of course, using normal pads, or using the FMP(4, 4) in a larger SoC (System-on-Chip), disregards this situation.

[1]It was claimed before (see Section 2.2) that power consumption could be reduced by reducing performance. However, caches are used to actually increase performance! In the FMP(4, 4) case, using caches may both increase performance and reduce power consumption!

Chapter 7

Conclusion

The past chapters described a multiprocessor architectural framework for increasing the lifetime reliability and fault tolerance against single event effects. Reducing power consumption and improving performance (especially for timing-critical systems) were also in the focus. The proposed mechanism introduced three basic operating modes of the multiprocessor: de-stress, fault-tolerant and high-performance, which could be dynamically changed according to the current application requirements. The main motivation behind this proposal was the fact that multiprocessor applications dynamically change their reliability and performance requirements. Thus, when high error resilience against SEEs is required, the application could put the multiprocessor in fault-tolerant mode. For high performance, where reliability requirements are low, the application could opt the high-performance mode. Finally, de-stress mode could be selected when the application performs low performance tasks that do not require high fault tolerance.

Several novel concepts like the low-complexity, self-calibrating aging monitors, the YFRR de-stress pattern, the programmable NMR voters, the recovery mechanism, were compiled to build the novel multiprocessor architectural framework. Besides novelties in the proposed solutions, this thesis brought novelties in their evaluation too. A simple analytical method based on the Weibul distribution was used to evaluate the RR and YFRR patterns of core gating. Furthermore, core-level NMR redundancy was investigated on a multiprocessor platform with 16 cores. The simulation results led to closed form expression that enables easily finding whether an NMR group of cores would survive the mission time without invoking recovery mechanisms, given the fault rate. This is appreciated by timing-critical, or, real-time systems. Of course, recovery mechanisms were also introduced and examined, since they would significantly improve the multiprocessor fault tolerance.

7.1 Are the objectives met?

The topic treated in the thesis is very wide and requires a lot of work, investigation, and experimenting. The work presented in the thesis was able to answer the basic postulates, but even more important, it was able to quantify them! That is, the

evaluation in Section 6.1 showed that operating the multiprocessor in de-stress mode which employs an YFRR gating pattern may prolong multiprocessor lifetime up to 31%. Further improvements are possible by using aging-aware task mapping and scheduling.

Moreover, the experiments conducted in fault-tolerant mode produced results that showed the correlation between the number of redundant modules in an NMR group of cores, the mission time and the fault rate (See Fig. 6.3, and Eqs. 6.9 and 6.10). Using Eq. 6.10 for example, one can quickly determine the number of cores that need to be arranged in an NMR group in order to accomplish a timing-critical task in a fault-tolerant manner. This quantification was done both for transient and permanent faults (Tables 6.4, 6.5 and 6.6). The performance overheads of state recovery mechanisms were also examined (Tables 6.7 and 6.8). At the end, simulations and measurements of power consumption (Tables 6.9, 6.10 and 6.11) were also made.

Thus, it can be said that most of the objectives stipulated in Subsection 1.4.3 are met. Two major challenges remain though. First, an investigation of the dynamical behaviour of the system using profiled applications and benchmarks of real-life systems would give the picture of the time that the multiprocessor spends in the three operating modes. Knowing the distribution of these periods will lead to deeper insights and improvements of the framework itself. Secondly, an investigation of the scalability of the proposed concept would give the answer whether the framework corresponds to the scale of the targeted problem.

7.2 Future work

Besides the two major challenges mentioned in Section 7.1, there are a lot of ideas, mechanisms investigations and experiments that may be implemented or conducted, or, are currently in the implementation phase. For example, a supplement to the PG and CG in de-stress mode could be DVFS. The cores could dynamically fine tune the scale of the voltage and frequency according to the performance requirements. Furthermore, temperature monitors could supplement the aging monitors, which would give a more precise picture of the age of the cores. As elaborated before, aging is strongly correlated with temperature.

7.2.1 Investigating aging effects and monitors

Special ICs that will be used to investigate the HCI and NBTI aging effects, as well as the operation of the aging monitors detailed in Subsection 4.1.2 are planned. The effect of core gating patterns with various duty cycles will be investigated. In this direction, a special environment for aging acceleration will be used. That is, aging effects could be accelerated significantly if the circuit is operated in an environment with abnormal temperatures and with supply voltage that is greater than the nominal (see Subsection 1.2.1). Both extremely low (-55°C) and extremely high (+200°C) temperatures will be used, which accelerate the HCI and the NBTI effects, respectively. Furthermore, aging as a function of the operating frequency will be also investigated.

7.2.2 Pushing the limits of core-level NMR

A general scheme for dynamic core-level NMR using a pool of cores and a pool of voters is feasible if the cores are identical and synchronous (driven by the same clock), which is the main motivation in [SKK14b]. Fig. 7.1 shows several possible formations of NMR groups in a 16-core multiprocessor. Note that the maximum number of voters required is $Q = P/2$, since maximum $P/2$ DMR systems may be formed in a multiprocessor with P cores. The example in Fig. 7.1 uses only three of the voters V_0, V_1 and V_2 in order to form the 4MR, TMR and DMR groups, respectively. The 4MR group is formed using C_0, C_1, C_4 and C_5, the TMR group is formed using C_6, C_7 and C_{10}, while the DMR uses C_9 and C_{12}. Note that voters V_3 to V_7 are not used in the current configuration, while C_2, C_3, C_8, C_{11}, C_{13}, C_{14} and C_{15} are used as independent, stand alone cores.

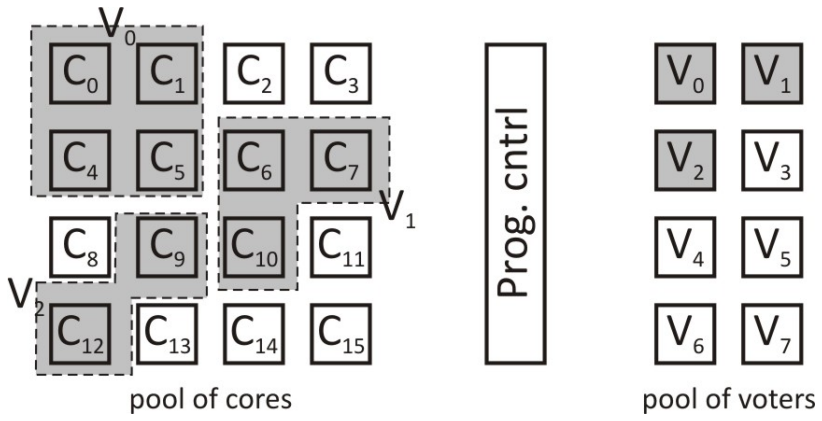


Figure 7.1: Dynamic core-level NMR group formation using programmable NMR voters

In order to form an arbitrary number of NMR groups, each with an arbitrary set of N cores, a simple programmable controller is needed (Prog. cntrl in Fig. 7.1), which could dynamically connect the cores to the voters. Upon an application request, the controller could reconfigure the system in another form in a few clock cycles. E.g., the multiprocessor configuration of Fig. 7.1 could be transformed to a single TMR group consisting of C_0, C_3, C_{15} and V_5, while the rest of the cores could be set as stand alone. Of course, the controller could be programmed to form no NMR groups at all, in which case the multiprocessor would operate as any regular multiprocessor (the voters are not used in this case). Another extreme is to put all 16 cores in one NMR group using a single voter.

In an NMR group, the outputs of all cores are actually inputs of the assigned programmable NMR voter. The N cores are tightly synchronized, executing simultaneously the same instruction, and are thus viewed by an external system as a single core. In each clock cycle the voter selects the actual output of the NMR group. In the example configuration of Fig. 7.1 the number of cores is effectively reduced to ten. The external system recognizes the seven stand alone cores, but considers that each of the three NMR groups is a single core.

In this scheme, not only the cores are redundant, but also are the voters. If the system perceives that a voter constantly reports erroneous self-checks, it will simply not use it any more. This scheme would enable binding any set of cores to any of the voters in the pool.

However, in order to relax the requirements that the cores in a core-level NMR system are tightly synchronous and homogeneous, a different approach is required. Initial considerations go in the direction that voting should be done only for the "store" operations, when the cores in the NMR group write into memory. Each write would be redirected to a special NMR voter which has to wait all active cores in the NMR group to supply the data. After voting, the voter itself is the actual entity that writes the (majority) result to memory.

This approach has several challenges arising from the fact that the cores do not supply the data synchronously. Firstly, the voters have to wait for data in a limited time period, since if a core fails to supply data due to an error, the system will be blocked. This time period is not necessarily deterministic. If a core fails to write data, or the data between cores differ, interrupts should be triggered. Secondly, the cores should get a feedback from the voter if their data is written to memory so they can continue processing, otherwise memory inconsistencies may occur. Both of these issues seriously affect the performance[1] and increase the complexity of the system.

7.2.3 Other considerations

At the application layer, an investigation of the mixed modes of operation shortly described in Subsection 4.3.1 may give valuable insights. E.g., de-stressing a NMR group of cores with a "fresh" NMR group of cores would provide fault tolerance, but also a time for recovery from effects like NBTI. However, the most significant extension of the application layer would be the introduction of lifetime-aware task mapping and scheduling, based on the information supplied by the aging monitors.

Finally, scalability regarding the number of cores in the framework requires examining many aspects related to the interconnection network, such as redundant links and fault-tolerant routing/switching protocols.

[1] As a guideline, around 12% of the program instructions are memory write operations [44].

Appendix A

A simple and flexible 32/64-bit RISC core

Multiprocessors are widely used in all computing segments i.e., desktop, server and embedded. The embedded domain has the most diverse application requirements regarding performance, power consumption and dependability. Therefore, a design of a core that is planned to be used in different embedded systems should enable an easy trade-off between these attributes. Of course, the cost is another system attribute that has to be taken into account. A specific and desirable property of embedded systems is predictable execution time.

Lots of IP (Intellectual Property) cores of (multi)processors are currently available, both commercial and non-commercial. The ARM [9] and ARC [112] microprocessors are widely used for embedded applications such as mobile and multimedia devices, game consoles, computer networking and communications, etc. These processors have modular designs, or different versions for specific usage like DSP (Digital Signal Processing) applications, or fault-tolerant mechanisms for use in space projects like the LEON processor [35]. However, these processors are designed for specific SoC applications with standard bus interfaces that enable communication between the different components in the SoC. On the other side, powerful general-purpose architectures like MIPS [80], SPARC [67], PowerPC [55] (and ARM in a way) although RISC, have relatively complex instruction set architectures (ISA). Some of them do not completely adhere to the RISC design principles (i.e., not pure RISCs) in order to boost performance for specific applications. Nonetheless, there is a lack of simple and easy-to-use processor cores for embedded multiprocessing environments and SoCs, especially for 64-bit systems. This is partially a motivation behind the OpenRISC [84], which is an open-source project for general-purpose 32- and 64-bit microprocessors with optional vector processing support, as well as the OpenSPARC [85] architecture.

The FMP(4, 4) designed to investigate the thesis proposal uses a novel load/store RISC core [Sim13] with flexible design space which offers simplicity and ease-of-use. First instance of flexibility is the core's operational width i.e., 64- or 32-bit. The instruction set is completely compatible between the two widths. Special features include: classical stack, novel interrupt organization and virtual memory system. Furthermore, if caches are not used (or are switched off), an exact prediction of the

execution time is possible. The core is designed in an environment for simulation, testing and co-verification of microprocessors presented in Section 5.5 (see also [SKK12]). The FMP(4, 4) is built of eight such cores (32-bits wide). It is successfully produced and tested in the IHP 130 nm technology, and used in many experiments aimed at evaluation of the thesis proposal.

A.1 Core architecture

The proposed load/store RISC core comes in two widths – 64- and 32-bit. The 64-bit version has a 64-bit data bus, 40-bit address bus and 64 (64-bit wide) General Purpose Registers (GPRs). The 32-bit version has a 32-bit data bus, 24-bit address bus and 32 GPRs (32-bit wide). Fig. A.1 shows the core's architecture.

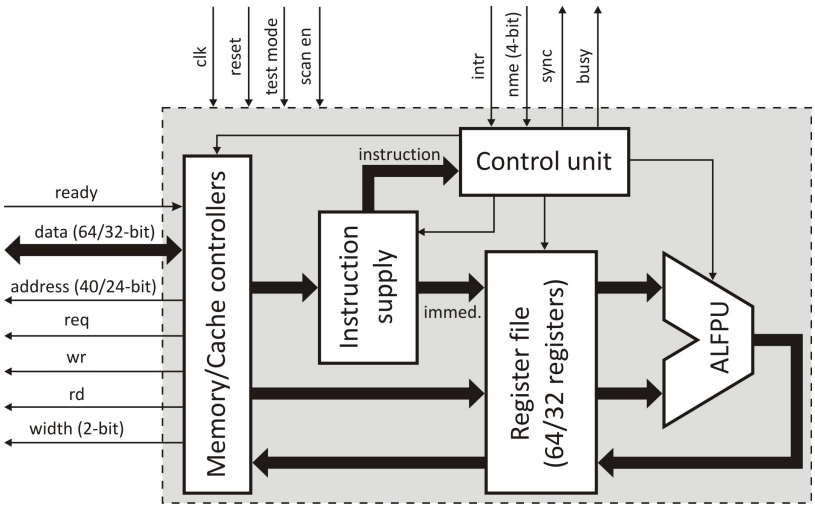


Figure A.1: A simple and flexible 64/32-bit RISC core

The thick arrows in Fig. A.1 depict the data-path, while the thin arrows show the control. The Instruction supply unit extends the signed/unsigned immediate values in instructions to 64/32 bits and passes them to the Register file or the Arithmetic/Logic and Floating-Point Unit (ALFPU). The ALFPU has two 64/32-bit inputs coming from the selected two registers in the register file. In the 8-core implementation the FPU part is not included, while the FPU support is left, so encountering a floating-point instruction will not raise exceptions but will also not change the microprocessor state – equivalent to executing a NOP (no operation) instruction. Although not shown (for the sake of clear presentation) in Fig. A.1, the second input of the ALFPU can be an immediate value from the Instruction supply block; Section A.2 shows that the second argument of the arithmetic/logic operations can be either a register or an immediate value.

Cache support

The core provides instruction support for L1 data and instruction caches and L2 shared cache. Inclusion of higher level caches is possible, although there is no explicit instruction support. Fig. A.2 presents a microprocessor based on the proposed core.

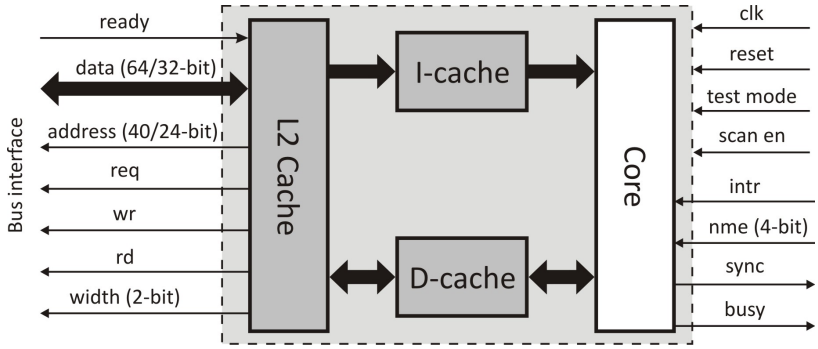


Figure A.2: A microprocessor with L1 and L2 caches

The designer is free to choose the inclusion/exclusion of caches as well as their size and number of blocks (of course, in a limited design space). For example, the cores in the 8-core implementation have only L1 caches.

Bus interface

Fig. A.3 shows the bus interface (BI) Finite State Machine (FSM) that consists of three states NT (no transfer), IF (instruction fetch) and RW (data read/write). The core does not fetch instructions i.e., it is in the NT state only in reset and wait states, otherwise it is in the IF state. When executing a load/store instruction, it transits to the RW state.

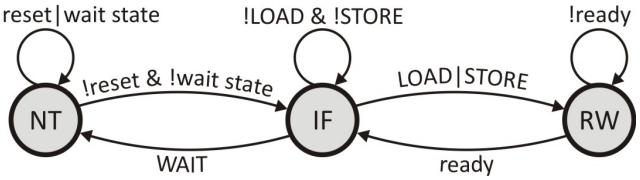


Figure A.3: Bus interface FSM

The memory and peripherals do not make a difference between an instruction fetch and data read. Furthermore, peripherals could be only memory-mapped, so the core does not make a difference whether the transfer is to/from memory or peripherals.

Fig. A.4 shows an example bus operation. Regarding the bus interface, the core operates similarly to a master device of an AMBA AHB 2.0 bus [8]. In the first cycle, the core requests a read operation setting the *address* and the transfer *width* (specifying byte, halfword, word or doubleword) and raising the signals *req* and *rd*. Note that the core supports only aligned memory access, e.g., a halfword read/write

at an odd address will raise an exception. The addressed unit sets the *ready* signal in the second cycle, placing the requested data on the *data* bus.

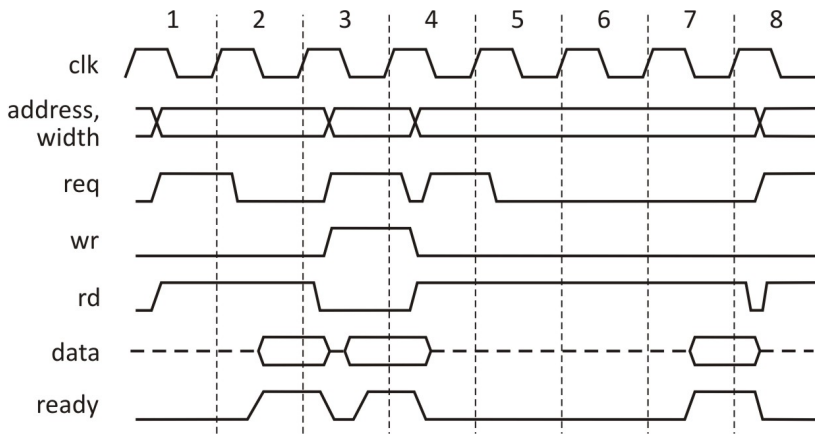


Figure A.4: Bus interface cycles

In the third cycle the core requests a write operation, raising *wr* and placing the data on the data bus. The addressed unit responds in the same cycle, raising *ready*. However, a transfer to/from slower units could take an arbitrary number of cycles. For example, the initiated transfer in the fourth cycle is completed in the seventh.

Data transfer addressing modes

The core provides three (virtually four) addressing modes for data transfers in order to support high-level programming languages. All GPRs could be used by load/store instructions to form the effective addresses.

Displacement. A base specified in a GPR and an immediate, 12-bit signed offset (contained in the load/store instruction) form the effective address. Thus, effective address = base + offset. For zero offset, the **register** addressing mode is obtained.

Indexed. Two GPRs specify the base and the index of an effective address i.e., effective address = base + index.

Immediate. The LOAD instruction has an immediate form, where an 18-bit signed/unsigned immediate value could be specified.

Execution FSM

Fig. A.5 shows the seven-states execution FSM of the core. The core is in the **Empty (E)** state during reset and when waiting for the BI to supply instructions. If instructions are ready for execution, the core transits to the **Fetch and execute (FEX)** state. An exception or interrupt (*excint* in Fig. A.5) puts the core into the **Exceptions and interrupt (INT)** state.

Simple instructions are executed in one cycle in the FEX state. Multi-cycle (*mc*) instructions have one FEX state and several **Execution (EXE)** states. Load/store instructions put the core into the **Data Transfer (DT)** state if the requested transfer

occurs within the current process; otherwise, the core has to check (in the **Shared data access check (SDC)** state) if the requested transfer is allowed. If so, the core transits to the DT state, otherwise an exception is invoked, transiting to the INT state.

If the current instruction is simple, the core checks for exceptions and interrupts before fetching a new instruction. If so, it transits to the INT state. If *excint* is inactive and a new instruction is ready to be fetched from the instruction supply (Fig. A.1) i.e., empty is inactive, the core stays in the FEX state, otherwise it transits to the E state.

The core stays in the EXE state till the *last* cycle of the multi-cycle instruction. Similarly like in the FEX state, it goes in the E state if the following instruction is not ready, otherwise it transits to the FEX state. If *excint* is active it goes to the INT state. The same holds when the core is in the DT state, where instead of the *last* cycle, the core waits the *ready* signal to acknowledge the transfer.

The WAIT instruction puts the core into the **WAIT** state. Only a hardware exception or a non-masked hardware interrupt could take the core out of this state i.e., into the INT state. Of course, active reset puts the core into the E state.

In the INT state the core behaves as follows. In the case of a hardware interrupt, the core has to read the interrupt specification from the interrupt controller. The interrupt controller is a peripheral that puts the specification on the data bus. Therefore, the core transits to the DT state. In a case of exception, the core has all data available and transits immediately to the E state, since the first instruction of the exception handler that resides in memory could not be loaded in this single cycle.

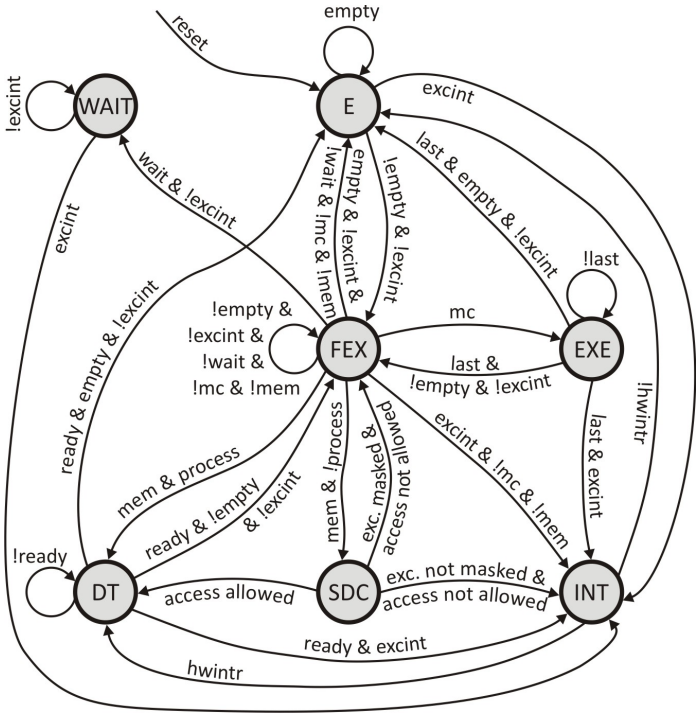


Figure A.5: Execution FSM

Exceptions and interrupts

The core has by default 4 hardware NMEs (Non-Maskable Exceptions) that are invoked by the *nme* input signal. The designer could actually specify the number of NMEs (from 0 to 8) by setting an appropriate parameter. NMEs signal critical situations that should be immediately treated.

Software exceptions are maskable and occur when instructions produce situations that require special attention. Table A.1 shows all software exceptions, ordered by priority.

Table A.1: Software exceptions

Exception	Remark
Invalid instruction	unknown instruction, wrong code
Protection violation	four types of violation
Unaligned memory access	address/data width is not aligned
Overflow/Underflow	the result does not fit into the GPR
Division by zero	the divisor operand is zero
Floating-point	invoked by FP ops, e.g., FP overflow
Timer/watchdog	a timer/watchdog reached zero

There are by default two *timers/watchdogs*, with a special exception for each of them. The designer could easily change the number of timers/watchdogs (from 0 to 8) and with that add/remove the appropriate exceptions for each timer/watchdog.

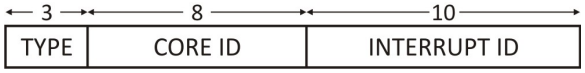


Figure A.6: Interrupt specification

Hardware interrupts are invoked by an interrupt controller over the *intr* input. The interrupt controller places the interrupt specification on the data bus when the core acknowledges the interrupt by initiating data read operation at the interrupt controller's address. Software interrupts could be *inter-processor* or *self-interrupts*. They are invoked by the INTR instruction. An inter-processor interrupt is sent to the interrupt controller that further interrupts the appropriate cores/processors. Fig. A.6 presents the hw/sw interrupt specification. The *Interrupt ID* specifies which interrupt to be invoked, while the *Core ID* specifies which core/processor to be interrupted. Table A.2 presents the types of interrupts.

If there is no core with the specified core ID, the interrupt is ignored. The type 001 specifies that the controller should interrupt an inactive core. For that purpose, the controller should observe the *busy* outputs of the cores in the system. If bit 4 of the control register is set, the core treats self-interrupts as inter-processor interrupts and sends the interrupt specification to the interrupt controller. Although the core ID is 8-bits, and the interrupt ID is 10-bits wide, the designer could easily change these widths. The total width including the 3-bits for type should not exceed 24.

Table A.2: HW/SW interrupt types

Type	Interrupt type
000	Self-interrupt (core ID is irrelevant)
001	Single interrupt to inactive core (if all active, to core ID)
010	Single interrupt, exclusively to core ID
011	Broadcast to all cores
100	Broadcast to all cores, except to core ID

Priority is resolved in the following manner. NMEs have highest priority (starting with NME 0). Then, exceptions follow with priorities as in Table A.1. Hardware interrupts have the lowest priority. The appropriate handler can define for itself whether it could be interrupted or not, by setting the appropriate masks and enable/disable interrupt bits.

A.2 Instruction set

All instructions are 32-bit wide with 0, 1, 2 or 3 arguments. The two most-significant bits (MSBs) of the instruction define the type of the instruction: load/store, arithmetic/logic/fp or control.

Data transfer instructions

Fig. A.7 shows the layout of the data transfer i.e., LOAD/STOR instructions. The D bit has to be 1. The L bit defines a LOAD (L=1) and a STOR (L=0) instruction. There are four types of load: with base+index (O=0) or base+offset (O=1) addressing, register-register (copy), and load immediate (18-bits wide). On the other hand there are two types of store: with base+index or base+offset addressing. The U-bit specifies whether the transfer refers to unsigned (U=1) or signed (U=0) data. The 12-bits wide offset is always signed.

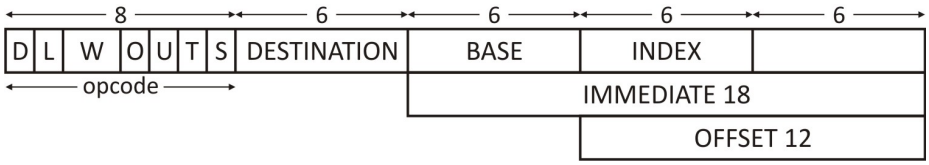


Figure A.7: Data transfer instructions

Table A.3 shows the data transfer types for LOAD and STOR instructions. The 2-bit W field specifies the data width of the transfer for T=0.

Register-register transfer could occur between GPRs, between system registers or between a GPR and a system register. For T=1 and W=10, the O-bit decides whether the destination is a GPR (O=0) or a system register (O=1). Similarly, the U bit decides the source (0 for GPR, 1 for system). The destination and source are

Table A.3: Data transfer types

W	T = 0	T = 1
00	64-bit	not used
01	32-bit	reserved for floating-point
10	16-bit	register-register
11	8-bit	load immediate

specified as 6-bit numbers. The source is specified in the *index* field in Fig. A.7. The S-bit is set (S=1) for LOAD/STOR instructions that involve a system register.

Arithmetic/logic and floating-point instructions

Fig. A.8 shows the layout of the AL and FP instructions. Here, D=0 and C=0. The first operand is a GPR specified by the 6-bit destination field. The second operand could be either a GPR specified by the 6-bit source (here, I=0), or an 15-bit immediate value (I=1). The result of multiplication or division (which is 128/64-bit in 64/32-bit core, respectively), is written into the two GPRs specified as destination and source. By division, the quotient and the reminder are placed in the specified destination and source GPRs, respectively. Therefore, the MUL (multiplication) and DIV (division) instructions do not have the "immediate value" form. The CONV (conversion from FP to integer or vice versa) instruction and all FP instructions also do not have an immediate form.

If the specified destination is equal to the specified source of a MUL operation, only the lower half of the result is written to the GPR. By DIV, only the quotient (with value 1) is written to the specified GPR.

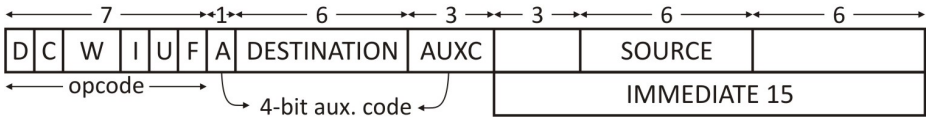


Figure A.8: Arithmetic/logic and floating-point instructions

The 2-bit W field specifies the width of operation with the same codes as in Table A.3 for T=0. Signed/unsigned integer operation is specified by U (0/1), while in FP operations the F-bit is set. Table A.4 shows the auxiliary (auxc) codes of all AL and FP instructions.

Standard operations of addition/subtraction and multiplication/division can be both integer or FP. ABS (FP absolute value), CMP (FP compare), NEG (FP negation) and SQRT (square root) are FP only. Shifts and rotates could be left (SL, RL) or right (SR, RR). By arithmetic right shift U=0 while by the simple right shift U=1. Also, by logic operations AND, NAND, OR and XOR the U bit is set.

Table A.4: 4-bit auxiliary code specifying AL and FP operations

Operation	Auxc[3:2]	F
ADD, SUB	00	0/1
MUL, DIV, CONV	01	0/1
RL, RR, SL, SR	10	0
AND, NAND, OR, XOR	11	0
ABS, CMP, NEG, SQRT	10	1

Control instructions

Fig. A.9 shows the layout of the control instructions. Here, D=0 and C=1.

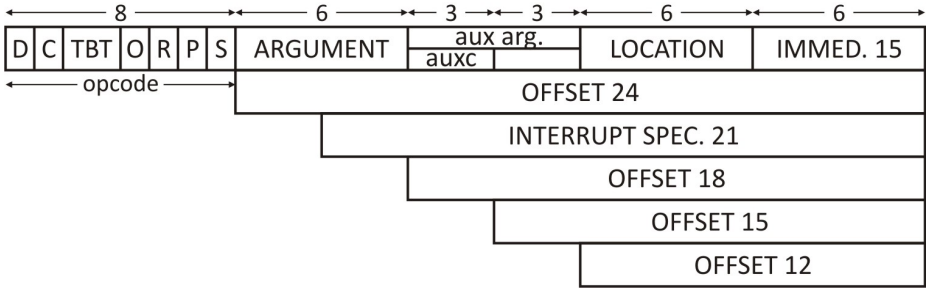


Figure A.9: Control instructions

The TBT field specifies the control Transfer or Branch Type, but is also used for other instructions. The O-bit defines whether the location of the (un)conditional control transfer is specified by an offset (O=1) or by GPR (O=0). The R-bit defines whether the transfer is IC-relative (R=0) or absolute (R=1). For absolute transfers O has to be 1, i.e., the absolute location has to be specified by a GPR. If the P bit (procedural transfer) is set, the current instruction pointer is saved on stack before the jump to the specified location. Tables A.5 and A.6 present the control transfer instructions.

Table A.5: Conditional and unconditional control transfer instructions

TBT	S	Instructions	Used fields
00	0	JMP	location/offset24
00	1	JMP.S	argument, aux. arg., location/offset12
01	0	Simple branches	argument, auxc, location/offset15
10	0	BE	argument, aux. arg., location/offset12
11	0	BNE	argument, aux. arg., location/offset/12

The JMP instruction performs an unconditional transfer to the specified location. The bit S is set (S=1) for system instructions, otherwise S=0. For example, a system jump (JMP.S) is used for context switch. Instructions in Table A.6, as well as BE (branch equal) and BNE (branch not equal) perform conditional transfers. BE and

Table A.6: Simple branches

auxc	Instruction	Branch if
000	B0	zero
001	BN0	not zero
010	BP	positive
011	BPZ	positive or zero
100	BN	negative
101	BNZ	negative or zero
110	BT_F	FP operation bit true
111	BF_F	FP operation bit false

BNE compare the values in the GPRs specified by the *argument* and the *auxiliary argument* and if the condition is satisfied, jump to the specified location.

Table A.7: Other control instructions

TBT	P	S = 0	S = 1
00	0	RET	RETE
00	1	PUSH	PUSH.S
01	0	POP	POP.S
01	1	/	/
10	0	INTR	WAIT
10	1	INTR	FLSH
11	0	/	SB
11	1	/	RB

Table A.8: FLSH immediate values

Immed. 6	FLSH
000001	SDAT
000010	NEI
000100	I Cache
001000	D Cache
010000	L2 Cache

However, if the instruction is not of the control transfer type, the O, R and P bits encode other states too. Table A.7 shows the rest of the control instructions.

Although the PUSH and POP instructions actually transfer data to/from the top of the stack, and more naturally belong to the data transfer group of instructions, they are put here since the code space is substantially exhausted for the data transfer instructions and a further extension of the instruction set would be harder. PUSH and POP use the *argument* field to specify a GPR (S=0) or a system register (S=1) as source/destination.

The RET instruction takes the top of the "normal" memory stack. The core resumes execution at the address fetched by RET. RETE takes the top value from a built-in-core NEI (exception/interrupt return address) stack. RET and RETE do not have arguments. The depth of the NEI stack is by default three, but the designer could easily change this parameter in the range from 3 to 16. The instruction counter is automatically pushed in the NEI stack in the INT state on each exception/interrupt.

The INTR instruction is used to invoke software interrupts, as explained in Subsection A.1. The interrupt specification could be specified by an immediate (21-bit) value or by a GPR.

The WAIT instruction puts the core in the WAIT state (Fig.A.5).

SB and RB set and reset single bits in the system registers. The system register is specified in the *argument* field, while the bit is specified in the *immed. 6* field.

The FLSH (flash/flush) instruction has several usages differentiated by *auxc*. The form which uses the *argument* field (auxc = 000) is used to fill the SDAT (shared data access table) of the novel virtual memory system. The *immed.6* form where auxc = 001 (shown in Table A.8) is used to flush the caches, the SDAT or the NEI. For example FLSH 28 (FLSH 0b011100) flushes all caches.

Assembly

Instruction assembly is proposed as follows. Firstly, the instruction mnemonic is specified and then the argument(s) (if any) separated by comma. The arguments could be GPRs (reg0, reg1,... reg31/reg63), system registers (srg0, srg1,..., srg18) or immediate values in binary, decimal or hexadecimal format. E.g., **ADD reg0, reg1** specifies that the result of adding the values in the GPR reg0 and the GPR reg1 is placed in reg0. **SUB reg3, 5** specifies that the result of subtracting the immediate value 5 from the value in reg3 is placed in reg3.

Load/store instructions have specific second argument. E.g., **LOAD reg2, reg3[5]** specifies displacement addressing with base in reg3 and displacement 5, while **LOAD reg2, reg7[reg4]** specifies indexed addressing with base in reg7 and index in reg4. The read data will be placed in reg2. If a system register is specified as destination (e.g., the control register) the form would be **LOAD.S CR, reg7[2]** or **LOAD.S srg16, reg7[2]**. The .S suffix is optional. An example load immediate would be **LOAD reg14, 345**.

64/32-bit compatibility

The instruction set is completely equivalent for both core versions. That is, software written for the 32-bit core could be executed on a 64-bit core, and vice versa. There are two precautions that have to be taken. Firstly, if a 64-bit program that uses the registers from 32 to 63, is executed on a 32-bit core, the 32-63 registers will be mapped in the range from 0 to 31 (since there are 32 registers in the 32-bit core). Therefore, if porting a 64-bit application to a 32-bit core is planned, it is best to avoid using the registers from 32 to 63, when possible. Furthermore, double word (64-bit) transfers

are not supported by the 32-bit version. If a load/store instruction specifies 64-bit transfer, a 32-bit core performs a 32-bit (word) transfer without raising exceptions.

Modes of operation and OS support

The control unit has 19 system registers specified in Table A.9. The core starts in *system mode* after reset. Resetting the 0-th bit of the control register CR (see Table A.10) switches the core to *user mode*. This could be done by executing the pseudo-instruction USRM (which translates to RB CR, 0). A form of the JMP.S instruction (JMP.S_U) is a convenient way for context-switches since it loads the process ID, the process base PB and length PL into the appropriate registers and additionally puts the core in user mode. Only an interrupt or exception can switch the core back to system mode. Thus, system calls should be implemented by invoking a self-interrupt with the INTR instruction.

The CR has many other functions such as enabling/disabling interrupts, caches and timers/watchdogs. The SYNC (SB CR, 8) and CSYN (RB CR, 8) instructions set the SYNC bit of the control register that directly drives the *sync* output of the core. This is used for synchronization purposes in a multiprocessor environment. The status of the NEI stack is automatically updated in bits 9 and 10. Write operations in the control register do not affect the NEI bits.

Table A.9: System registers

Nr.	Abbrev.	Register name
0	PB	Process Base
1	PL	Process Length
2	SP	Stack Pointer
3	EM	Exception Masks
4	IB	Interrupt handler Base address and process number
5	IL	Interrupt handler Length
6	NB	NME handler Base address and process number
7	NL	NME handler Length
8	EB	Exceptions handler Base address and process number
9	EL	Exceptions handler Length
10	EXC	Exceptions register
11	LI	Last Instruction before exception
12	IID	Interrupt ID register
13	AV	Address violation register
14	CID	Core ID register
15	ICA	Interrupt Controller's physical Address
16	CR	Control Register
17	TM0	Timer/watchdog 0
18	TM1	Timer/watchdog 1

The stack pointer SP points to the top of the stack and besides the use with PUSH and POP can be used to automatically save the Instruction Counter IC when executing a control transfer instruction (as explained in Subsection A.2).

Exceptions and interrupts' handler addresses, process numbers and lengths are specified in the IB, IL, NB, NL, EB, and EL registers. Exception masks could be set in the EM register, while triggered exceptions could be viewed and reset in the EXC register. The last instruction before the triggered interrupt/exception is saved into the LI register. The core loads the IID register after received interrupt on the *intr* line from the Interrupt Controller's address specified in ICA.

The CID register holds the core ID which is mainly used by the interrupt system, but is also very convenient in multiprocessor environments.

TM0 and TM1 registers hold the current count of the appropriate timers/watchdogs. If active, these values are decremented in each clock cycle. Once the timers/watchdogs reach zero, they trigger an exception.

Table A.10: Control register

Bit	Function	Value after reset
0	System mode	1
1	Enable hw interrupts	0
2	Enable self-interrupts	1
3	Enable inter-processor interrupts	1
4	Treat self-interrupts as inter-processor	0
5	Instruction cache active	0
6	Data cache active	0
7	L2 cache active	0
8	SYNC mode	0
9	NEI stack empty (not affected by write op)	1
10	NEI stack full (not affected by write op)	0
24	Timer/watchdog 0 active	0
25	Timer/watchdog 1 active	0

The control unit triggers a protection violation exception in the following cases: when it encounters a system instruction (e.g., write/read operations to/from the system registers) in user mode, when the instruction or data is out of process bounds, or when the data access is not allowed (the AV register records the violated address).

At last, it is worth mentioning that the core satisfies the classic virtualization requirements (i.e., trap-and-emulate) defined by Popek and Goldberg in [90].

A.3 Core performance evaluation

Since the appropriate C compiler is under construction, the performance of the core is evaluated with non-standard benchmark programs, written entirely in the assembly language. Actually, an instruction generator is used to generate around 128 K instructions. The number of clock cycles needed to execute the generated instructions

is evaluated without using any cache, and using a 64 KB directly-mapped instruction cache with 512 B block-size. Furthermore, in the case with L1 cache, two sub-cases are evaluated. Firstly, an array with $\sim$ 16 K instructions (that fits completely in the cache) loops 4 times. Then, an array of 128 instructions (that fills a single cache block) loops 1000 times. Table A.11 shows the average CPI (clocks per instruction) and the inverse IPC (instructions per clock). CPI/IPC are the usual figures for comparing processor architectures [44].

Table A.11: Core performance evaluation

	No cache	Loop in L1	Loop in L1 block
Nr. instructions	128.001	130.386	128.002
Nr. clocks	602.890	296.749	216.846
CPI	4,71	2,27	1,69
IPC	0,21	0,44	0,59

Appendix B

Library of framework middleware procedures

Table B.1: Library of FM procedures. The procedures are presented in C-like style. `*int` denotes array of integers.

Procedure	**Description**
Command	
`void enableActions()`	reset DA bit
`void unfreezeLastActions()`	reset FLAR bit
`void globalReset()`	set GR bit
`void resetModules(*int)`	set RM_i bit(s)
`void resetErrorCounters(*int)`	set REC_i bit(s)
`void resetErrorCountersAll()`	set all REC bit(s)
`*int getAge(*int)`	Initiate age read-out (set AGE_i bit(s)) and read aging monitor(s)
PG/CG	
`int getPGCG()`	get contents of PG/CG register
`void setPGCG(int)`	set contents of PG/CG register
`void powerOff(*int)`	power-off module(s)
`void powerOn(*int)`	power-on modules(s)
`void clockOff(*int)`	clock-off module(s)
`void clockOn(*int)`	clock-on module(s)
`void activate(*int)`	power-on and clock-on module(s)
`*int getPGoff()`	get a list of powered-off modules
`*int getPGon()`	get a list of powered-on modules
`*int getCGoff()`	get a list of clocked-off modules
`*int getCGon()`	get a list of clocked-on modules
`*int getActiveModules()`	get a list of active modules
`*int getInactiveModules()`	get a list of inactive modules
`int nrActive()`	get the number of active modules
`int nrInactive()`	get the number of inactive modules

`bool ifActive(int)`	check if module is active
`bool ifInactive(int)`	check if module is inactive
`bool ifActiveAll()`	check if all modules are active
`bool ifOneActive()`	check if only one module is active
`bool ifPGoff(int)`	check if module is powered-off
`bool ifPGon(int)`	check if module is powered-on
`bool ifCGoff(int)`	check if module is clocked-off
`bool ifCGon(int)`	check if module is clocked-on
Mode	
`int getMode()`	get contents of mode register
`void setMode(int)`	set contents of mode register
`void formNMR(*int)`	form a NMR group of specified modules
`void noNMR()`	dissolve any NMR group formed
`void setAIO()`	set AIO bit
`void clearAIO()`	reset AIO bit
`void setIOMOD(int)`	set IOMOD field
`int getIOMOD()`	get IOMOD field
`bool ifAIO()`	check if AIO is set
`bool ifInNMR(int)`	check if module belongs to a NMR group
`bool ifDriver(int)`	check if module is specified in IOMOD
De-stress timer	
`void setDeStressPeriod(int)`	set contents of De-stress timer register
`int getDeStressTimerValue()`	get contents of De-stress timer register
`void noDeStress()`	set zero in De-stress timer register
`bool ifDeStressing()`	check if De-stress timer value > 0
`bool ifCouldDeStress()`	inverted `ifActiveAll()` function
Action registers	
`int getAction(int)`	get contents of specified action register
`*int getActionAll()`	get contents of all action registers
`void setAction(int, int)`	set specified action in specified register
`void setActionDA(int)`	set DA bit in specified register
`void clearActionDA(int)`	reset DA bit in specified register
`void setActionFLAR(int)`	set FLAR bit in specified register
`void clearActionFLAR(int)`	reset FLAR bit in specified register
`void setAction(int, bool, bool, int, int, int, *int)`	set specified action in specified register, field by field
`void setNoAction(int)`	set no action in specified register
`void setNoActionAll()`	set no action in all registers
`void setActionOUTDRV(int, int)`	set OUTDRV field in specified register
`void setActionACTION(int, int)`	set ACTION field in specified register
`void setActionARG0(int, int)`	set ARG0 field in specified register
`void setActionARG1(int, *int)`	set ARG1 field in specified register

`void ifActionDA(int)`	check if DA action is set
`void ifActionFLAR(int)`	check if FLAR action is set
`void ifNoAction(int)`	check if no action is set
`void ifActionIntr(int)`	check if action is an interrupt
`void ifActionReset(int)`	check if action is a reset
Error counters	
`int getErrorCount(int)`	get contents of specified error counter
`*int getErrorCountAll()`	get contents of all error counters
`bool ifErrorCount(int)`	check if errors occurred in module
`bool ifErrorCountAll()`	check if errors occurred at all
Error timers	
`void setErrorPeriod(int, int)`	set contents of specified error timer
`void setErrorPeriodAll(int)`	set contents of all error timers
`int getErrorTimerValue(int)`	get contents of error timer
`*int getErrorTimerValueAll()`	get contents of all error timers
`void stopErrorTimer(int)`	set zero in specified error timer
`void stopErrorTimerAll()`	set zero in all error timers
Last action	
`int getLastAction()`	get contents of last action register
`bool ifFalseSetup()`	check if FSET bit is set
`bool ifVoterError()`	check if VERR bit is set
`bool ifGlobalReset()`	check if GRES bit is set
`bool ifModuleReset(int)`	check if RES_i bit is set
`bool ifModuleIntr(int)`	check if INT_i bit is set
`bool ifModuleError(int)`	check if ERR_i bit is set
`bool ifVoterIndecisive()`	check if IND bit is set
`int getNrDiff()`	get contents of NR_DIFF field
`bool ifError()`	check if at least one ERR_i bit is set
`bool ifIntr()`	check if at least one INT_i bit is set
`bool ifReset()`	check if at least one RES_i bit is set
`bool ifLastActionOK()`	check if last action register is zero
Interrupt status	
`int getIntrStatus()`	get contents of interrupt status register
`bool ifIntrPending()`	check if interrupt status is not zero
`bool ifVEI()`	check if VEI bit is set
`bool ifDTI()`	check if DTI bit is set
`bool ifOLI(int)`	check if OLI bit is set
`bool ifILI(int)`	check if ILI bit is set
`bool ifETI(int)`	check if ETI bit is set
`bool ifECI(int)`	check if ECI bit is set
`bool ifOLIAll()`	check if at least one OLI bit is set
`bool ifILIAll()`	check if at least one ILI bit is set

`bool ifETIAll()`	check if at least one ETI bit is set
`bool ifECIAll()`	check if at least one ECI bit is set
`void ackIntr(int)`	reset specified interrupt bit (acknowledge)
`void ackIntrAll()`	acknowledge all interrupts (reset to zero)
Interrupt mask	
`int getIntrMask()`	get contents of interrupt mask register
`void setIntrMask(int)`	set contents of interrupt mask register
`void enableIntrAll()`	enable all interrupts
`void disableIntrAll()`	disable all interrupts
`void enableVEI()`	enable VEI interrupt
`void disableVEI()`	disable VEI interrupt
`void enableDTI()`	enable DTI interrupt
`void disableDTI()`	disable DTI interrupt
`void enableOLI(int)`	enable OLI_i interrupt
`void disableOLI(int)`	disable OLI_i interrupt
`void enableOLIAll()`	enable all OLI interrupts
`void disableOLIAll()`	disable all OLI interrupts
`void enableILI(int)`	enable ILI_i interrupt
`void disableILI(int)`	disable ILI_i interrupt
`void enableILIAll()`	enable all ILI interrupts
`void disableILIAll()`	disable all ILI interrupts
`void enableETI(int)`	enable ETI_i interrupt
`void disableETI(int)`	disable ETI_i interrupt
`void enableETIAll()`	enable all ETI interrupts
`void disableETIAll()`	disable all ETI interrupts
`void enableECI(int)`	enable ECI_i interrupt
`void disableECI(int)`	disable ECI_i interrupt
`void enableECIAll()`	enable all ECI interrupts
`void disableECIAll()`	disable all ECI interrupts
Predefined outputs	
`int getPredefOutputs()`	get contents of predefined outputs register
`void setPredefOutputs()`	set contents of predefined outputs register
Inactive outputs	
`int getInactiveOutputs()`	get contents of inactive outputs register
`void setInactiveOutputs()`	set contents of inactive outputs register
YFRR support	
`*int youngestFirst(*int)`	sort input modules by age (youngest first)
`*int oldestFirst(*int)`	sort input modules by age (oldest first)

List of Abbreviations

ALFPU	Arithmetic/logic and floating-point unit, page 138
ALU	Arithmetic/logic unit, page 46
ARQ	Automatic repeat request, page 33
ASIC	Application-specific integrated circuit, page 8
BCH	Bose-Chaudhuri-Hocquenghem, page 32
BiCMOS	Bipolar-CMOS, page 8
BIST	Built-in self test, page 29
BTI	Bias temperature instability, page 11
BTU	Brandenburgische technische Universität, page ix
CDF	Cumulative distribution function, page 5
CG	Clock gating, page 60
CMOS	Complementary metal oxide semiconductor, page 8
COTS	Commercial off-the-shelf, page 40
CPI	Clocks per instruction, page 99
CRR	Checkpointing, rollback and retry, page 33
CRV	Constrained-random verification, page 112
DDR	Double data rate, page 36
DMR	Dual-modular redundant, page 24
DVFS	Dynamic voltage and frequency scaling, page 17
ECC	Error correction codes, page 30
ECL	Emitter-coupled logic, page 8

EDAC Error detection and correction, page 70

EDP Energy-delay product, page 49

EDUP Energy-delay-upset rate product, page 51

EMI Electromagnetic interference, page 10

EOS Electrical overstress, page 10

ESD Electrostatic discharge, page 10

FC Framework controller, page 59

FEC Forward error correction, page 30

FEEIT Faculty of electrical engineering and information technologies, page ix

FIL Fault injection logic, page 101

FinFET Fin-shaped field effect transistor, page 19

FIT Failures in time, page 7

FM Framework middleware, page 59

FMP Framed multiprocessor, page 59

FPGA Field-programmable gate array, page 8

FPSR Field-programmable self-repair, page 52

FSM Finite state machine, page 21

FWG Framework group, page 59

GPR General-purpose register, page 138

GPU Graphic processing unit, page 22

HBD Hard breakdown, page 13

HCI Hot carrier injection, page 11

HDL Hardware description language, page 113

HKMG High-K metal gate, page 12

I/O Input/output, page 34

IC Integrated circuit, page 3

IGS International graduate school, page ix

IHP Institut für Halbleiterphysik, page ix

ILP Instruction-level parallelism, page 21

IML Input multiplexing logic, page 71

IP Intellectual property, page 137

IPC Instructions per clock, page 99

ISD Input state descriptor, page 68

ISS Instruction set simulator, page 113

ITRS International technology roadmap for semiconductors, page 16

L1 Level 1, page 18

LEO Low Earth orbit, page 13

LER Line edge roughness, page 9

LWR Line width roughness, page 9

MBU Multiple bit upset, page 15

MCCP Multiple clustered core processor, page 52

MOSFET Metal oxide semiconductor field effect transistor, page 15

MPSoC Multiprocessor system-on-chip, page 50

MTBF Mean time between failures, page 6

MTTF Mean time to failure, page 6

MTTR Mean time to repair, page 6

NASA National aeronautics and space administration, page 3

NBTI Negative BTI, page 12

NMOS Negative-channel metal oxide semiconductor, page 11

NMR N-modular redundant, page 24

NMROD NMR on demand, page 24

NoC Network-on-chip, page 47

OML Output multiplexing logic, page 71

PBD Progressive breakdown, page 13

PBTI Positive BTI, page 12

PCB Printed circuit board, page 19

PDF Probability density function, page 5

PDP Power-delay product, page 49

PE Processing element, page 21

PG Power gating, page 61

PMOS Positive-channel metal oxide semiconductor, page 11

PNPN Positive-negative-positive-negative, page 15

RAM Random access memory, page 20

RAMP Reliability aware microprocessor, page 50

RAS Reliability, availability and serviceability, page 34

RDF Random dopant fluctuations, page 9

RISC Reduced instruction set computer, page 40

RR Round-robin, page 61

RTL Register-transfer level, page 112

SBD Soft breakdown, page 13

SDF Standard delay format, page 102

SEB Single event burnout, page 15

SEC Single error correction, page 31

SEC-DED Single error correction – double error detection, page 31

SEDR Single event dielectric rupture, page 15

SEE Single event effect, page 10

SEFI Single event functional interrupt, page 15

SEGR Single event gate rupture, page 15

SEL Single event latch-up, page 15

SER Soft error rate, page 20

SET Single event transient, page 15

SEU	Single event upset, page 14
SMT	Simultaneous multi-threading, page 39
SoC	System-on-chip, page 132
SRAM	Static random access memory, page 34
SSN	Simultaneous switching noise, page 10
TDDB	Time-dependent dielectric breakdown, page 11
TeV	Tera electron Volt, page 14
TID	Total ionizing dose, page 10
TML	Test, monitor and log, page 114
TMR	Triple-modular redundant, page 24
TTF	Time to failure, page 5
TTR	Time to repair, page 6
VLIW	Very long instruction word, page 50
VLSI	Very large scale of integration, page 17
YFRR	Youngest first round-robin, page 63
ZUSYS	Zuverlässige Systeme, page ix

Bibliography

[1] Methods for calculating failure rates in units of fits. *JEDEC Standard No. JESD85*, pages 1–22, 2001.

[2] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Isolation in Commodity Multicore Processors. *Computer*, 40(6):49–59, 2007.

[3] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, S. Pastore, and G.R. Sechi. Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, pages 105 –113, sept. 2007.

[4] H. Ando, R. Kan, Y. Tosaka, Keiji Takahisa, and K. Hatanaka. Validation of hardware error recovery mechanisms for the SPARC64 V microprocessor. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 62–69, June 2008.

[5] J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *Computers, IEEE Transactions on*, 42(8):913 –923, aug 1993.

[6] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, February 1990.

[7] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Guenther H. Leber. Comparison of Physical and Software-Implemented Fault Injection Techniques. *IEEE Transactions on Computers*, 52:1115–1133, 2003.

[8] ARM. AMBA Specification and Multi layer AHB Specification (rev2.0). www.arm.com, 2001.

[9] ARM. ARM Architecture Reference. http://www.arm.com, 2010.

[10] ARM. *Cortex-R4 and Cortex-R4F Technical Reference Manual*. ARM, 2011.

[11] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. IBM POWER7 systems. *IBM Journal of Research and Development*, 55(3):2:1–2:13, May 2011.

[12] M. Augustin, M. Goessel, and R. Kraemer. Reducing the area overhead of TMR-systems by protecting specific signals. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 268 –273, july 2010.

[13] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. 2001.

[14] K.J. Balakrishnan, G. Giles, and J. Wingfield. Test Access Mechanism in the Quad-Core AMD Opteron Microprocessor. *Design Test of Computers, IEEE*, 26(1):52–59, Jan 2009.

[15] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *In Proc. of the Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 170–177. ACM Press, 2003.

[16] R.P. Bastos, F.L. Kastensmidt, and R. Reis. Design of a robust 8-bit microprocessor to soft errors. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, page 2 pp., 0-0 2006.

[17] Thomas Baumann, Georg Georgakos, Christian Pacha, and Anselme Urlick Tchegho Kamgaing. Circuit arrangement with a test circuit and a reference circuit and corresponding method, August 2010.

[18] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli. Run-time mapping for reliable many-cores based on energy/performance trade-offs. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*, pages 58–64, Oct 2013.

[19] A. Bondavalli, F. Di Giandomenico, F. Grandoni, D. Powell, and C. Rabejac. State restoration in a COTS-based N-modular architecture. In *Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98) Proceedings. 1998 First International Symposium on*, pages 174 –183, apr 1998.

[20] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, Jul 1999.

[21] Sanghoan Chang and Gwan Choi. Gate-Level Exception Handling Design for Noise Reduction in High-Speed VLSI Circuits. In *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pages 109–114, Jan 2007.

[22] You-Sung Chang, Seungjong Lee, In-Cheol Park, and Chong-Min Kyung. Verification of a microprocessor using real world applications. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 181 –184, 1999.

[23] Yung-Yuan Chen, Shi-Jinn Horng, and Hung-Chuan Lai. An integrated fault-tolerant design framework for VLIW processors. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 555 – 562, 3-5 2003.

[24] Chen-Ling Chou and R. Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[25] Grzegorz Cieslewski, Alan D. George, and Adam Jacobs. Acceleration of FPGA Fault Injection Through Multi-Bit Testing. In Toomas P. Plaks, David Andrews, Ronald F. DeMara, Herman Lam, Jooheung Lee, Christian Plessl, and Greg Stitt, editors, *ERSA*, pages 218–224. CSREA Press, 2010.

[26] A. Das, A. Kumar, and B. Veeravalli. Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10, Sept 2013.

[27] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 689–694, March 2013.

[28] B. S. Dhillon. *Design Reliability - Fundamentals and Applications*. CRC Press, June 1999.

[29] A. Dixit and Alan Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 5B.4.1–5B.4.7, April 2011.

[30] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, Dec 2003.

[31] Evolven. Downtime, Outages and Failures - Understanding Their True Costs. http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html, 2013.

[32] Etienne Faure, Mounir Benabdenbi, and Francois Pecheux. Distributed online software monitoring of manycore architectures. In *Proceedings of the 16th IEEE International On-Line Testing Symposium (IOLTS'10)*, pages 56–61, 2010.

[33] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Maestro: Orchestrating Lifetime Reliability in Chip Multiprocessors. pages 186–200, 2010.

[34] M. Floyd, M. Ware, K. Rajamani, T. Gloekler, B. Brock, P. Bose, A. Buyuktosunoglu, J.C. Rubio, B. Schubert, B. Spruth, J.A. Tierno, and L. Pesantez. Adaptive energy-management features of the IBM POWER7 chip. *IBM Journal of Research and Development*, 55(3):8:1–8:18, May 2011.

[35] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 409 – 415, 2002.

[36] N. Gaitanis. The design of totally self-checking TMR fault-tolerant systems. *Computers, IEEE Transactions on*, 37(11):1450 –1454, nov 1988.

[37] Ian Glover and Peter Grant. *Digital Communications*. Prentice Hall, 1998.

[38] Rui Gong, Kui Dai, and Zhiying Wang. Transient Fault Tolerance on Chip Multiprocessor Based on Dual and Triple Core Redundancy. In *Dependable Computing, 2008. PRDC '08. 14th IEEE Pacific Rim International Symposium on*, pages 273–280, Dec 2008.

[39] T. Grasser, B. Kaczer, W. Goes, T. Aichinger, P. Hehenberger, and M. Nelhiebel. A two-stage model for negative bias temperature instability. In *Reliability Physics Symposium, 2009 IEEE International*, pages 33–44, April 2009.

[40] T. Grasser, B. Kaczer, W. Goes, H. Reisinger, T. Aichinger, P. Hehenberger, P. J Wagner, F. Schanovsky, J. Franco, P. Roussel, and M. Nelhiebel. Recent advances in understanding the bias temperature instability. In *Electron Devices Meeting (IEDM), 2010 IEEE International*, pages 4.4.1–4.4.4, Dec 2010.

[41] R.W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal, The*, 29(2):147–160, April 1950.

[42] A.S. Hartman, D.E. Thomas, and B.H. Meyer. A case for lifetime-aware task mapping in embedded chip multiprocessors. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 145–154, Oct 2010.

[43] Daniel Henderson and Jim Mitchell. POWER7 System RAS - Key Aspects of Power Systems Reliability, Availability, and Serviceability. White paper, 2012.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitive Approach*. Morgan Kaufmann (imprint of Elsevier), 4th edition, 2007.

[45] O. Heron, J. Guilhemsang, N. Ventroux, and A. Giulieri. Analysis of on-line self-testing policies for real-time embedded multiprocessors in DSM technologies. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 49–55, July 2010.

[46] Christian J. Hescott, Drew C. Ness, and David J. Lilja. A Methodology for Stochastic Fault Simulation in VLSI Processor Architectures. 2005.

[47] Eugene R. Hnatek. *Practical Reliability Of Electronic Equipment And Products*. CRC Press, October 2002.

[48] L. Hoffmann, G. Gardner, J. Lintz, J. Samson, C. Kouba, and R. Some. Designing the dependable multiprocessor space experiment. In *Radiation and Its Effects on Components and Systems, 2007. RADECS 2007. 9th European Conference on*, pages 1 –9, sept. 2007.

[49] Masashi Horiguchi and Kiyoo Itoh. *Nanoscale Memory Repair*. Springer, January 2011.

[50] M. Horowitz. Scaling, Power and the Future of CMOS. In *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pages 23–23, Jan 2007.

[51] Lin Huang, Feng Yuan, and Qiang Xu. Lifetime reliability-aware task allocation and scheduling for MPSoC platforms. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 51–56, April 2009.

[52] Lin Huang, Feng Yuan, and Qiang Xu. On Task Allocation and Scheduling for Lifetime Extension of Platform-Based MPSoC Designs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(12):2088–2099, Dec 2011.

[53] C.A. Hulme, H.H. Loomis, A.A. Ross, and Rong Yuan. Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing. In *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, volume 4, pages 2269 – 2276 Vol.4, 6-13 2004.

[54] A.S. Hwang. Radiation hardened 32-bit RISC microprocessor. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 5, pages 219 –226 vol.5, 2000.

[55] IBM. *PowerPC User Instruction Set Architecture*. IBM, January 2005.

[56] ITRS. Yield enhancement. www.itrs.net/Links/2007ITRS, 2007.

[57] Jianhui Jiang, Hongbao Shi, and Xiaodong Zhao. A novel NMR structure with concurrent output error location capability. In *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, pages 32 –39, 1999.

[58] Yong-Kyu Jung. Non-FPGA-based Field-programmable Self-repairable (FPSR) Microarchitecture. In *Adaptive Hardware and Systems, 2008. AHS '08. NASA/ESA Conference on*, pages 93 –100, june 2008.

[59] David Kammler, Junqing Guan, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A fast and flexible Platform for Fault Injection and Evaluation in Verilog-based Simulations. In *Secure Software Integration and Reliability Improvement (SSIRI), 2009, Proceedings of the IEEE International Conference on*, pages 309–314, Shanghai, China, jul 2009.

[60] Jagrit Kathuria, M. Ayoubkhan, and Arti Noor. A Review of Clock Gating Techniques. *MIT International Journal of Electronics and Communication Engineering*, 1, August 2011.

[61] T. Koal and H.T. Vierhaus. Optimal spare utilization for reliability and mean lifetime improvement of logic built-in self-repair. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 219–224, April 2011.

[62] Tobias Koal and Heinrich T. Vierhaus. Combining De-Stressing and Self Repair for Long-Term Dependable Systems. *Proceedings of the IEEE DDECS 2010, Vienna*, 2010.

[63] Cheng-Kok Koh, Weng-Fai Wong, Yiran Chen, and Hai Li. The salvage cache: A fault-tolerant cache architecture for next-generation memory technologies. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 268 –274, 4-7 2009.

[64] E. Kolonis, M. Nicolaidis, D. Gizopoulos, M. Psarakis, J.H. Collet, and P. Zajac. Enhanced self-configurability and yield in multicore grids. In *Proceedings of the 15th IEEE International On-Line Testing Symposium (IOLTS'09)*, pages 75–80, June 2009.

[65] Dongwoo Lee and Jongwhoa Na. A Novel Simulation Fault Injection Method for Dependability Analysis. *IEEE Design and Test of Computers*, 26:50–61, 2009.

[66] Kab Joo Lee and G. Choi. Design of a fault-tolerant microprocessor: a simulation approach. In *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, pages 161 –166, 15-16 1997.

[67] A.S. Leon, K.W. Tam, J.L. Shin, D. Weisner, and F. Schumacher. A Power-Efficient High-Throughput 32-Thread SPARC Processor. *Solid-State Circuits, IEEE Journal of*, 42(1):7–16, 2007.

[68] L.L. Lewyn. Physical design and reliability issues in nanoscale analog CMOS technologies. In *NORCHIP, 2009*, pages 1–10, Nov 2009.

[69] E. Litvinova, K. Mostovaya, and K. Krasnoyarujskaya. Fault diagnosis and repair of SoC memory. In *Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2008 Proceedings of International Conference on*, pages 635–639, Feb 2008.

[70] Shyue-Kung Lu, Huan-Hua Huang, Jiun-Lang Huang, and P. Ning. Synergistic Reliability and Yield Enhancement Techniques for Embedded SRAMs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(1):165–169, Jan 2013.

[71] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic Power Gating with Quality Guarantees. In *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 377–382, New York, NY, USA, 2009. ACM.

[72] S. Makar, T. Altinis, N. Patkar, and J. Wu. Testing of Vega2, a chip multiprocessor with spare processors. In *Test Conference, 2007. ITC 2007. IEEE International*, pages 1–10, Oct 2007.

[73] E. Maricau and G. Gielen. Computer-Aided Analog Circuit Design for Reliability in Nanometer CMOS. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 1(1):50–58, March 2011.

[74] E. Maricau and G. Gielen. *Analog IC Reliability in Nanometer CMOS*. Springer, 2013. ISBN: 978-1-4614-6162-3.

[75] Ritesh Mastipuram and Edwin C. Wee. Soft errors' impact on system reliability. www.edn.com, 2004.

[76] D.G. Mavis and P.H. Eaton. Temporally redundant latch for preventing single event disruptions in sequential integrated circuits, October 3 2000. US Patent 6,127,864.

[77] Carl D. Meyer. *Matrix analysis and applied linear algebra*. SIAM, April 2000.

[78] Timothy Miller, Nagarjuna Surapaneni, Radu Teodorescu, and Joanne Degroat. Flexible Redundancy in Robust Processor Architecture. 2009.

[79] E. Mintarno, J. Skaf, Rui Zheng, J. Velamala, Yu Cao, S. Boyd, R.W. Dutton, and S. Mitra. Optimized self-tuning for circuit aging. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 586 –591, 8-12 2010.

[80] MIPS. MIPS Technologies. www.mips.com, 2012.

[81] J.M. Mogollon, H. Guzman-Miranda, J. Napoles, J. Barrientos, and M.A. Aguirre. FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. In *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*, pages 169 –174, sept. 2011.

[82] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, Sept 2006.

[83] Michael Nicolaidis. *Soft Errors in Modern Electronic Systems*. Springer, 2010.

[84] OpenRISC. OpenRISC 1000 Architecture Manual. www.opencores.org, 2007.

[85] Oracle. OpenSPARC T1 and T2 architecture specification. www.oracle.com, 2008.

[86] Sangwoo Pae, A. Ashok, Jingyoo Choi, T. Ghani, Jun He, Seok hee Lee, K. Lemay, M. Liu, R. Lu, P. Packan, C. Parker, R. Purser, A. St.Amour, and B. Woolery. Reliability characterization of 32nm high-K and Metal-Gate logic transistor technology. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pages 287–292, May 2010.

[87] F. Paterna, L. Benini, A. Acquaviva, F. Papariello, A. Acquaviva, and M. Olivieri. Adaptive idleness distribution for non-uniform aging tolerance in MultiProcessor Systems-on-Chip. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 906–909, April 2009.

[88] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Critical Variable Recomputation for Transient Error Detection. 2007.

[89] Vladimir Petrovic. *Design Methodology for highly Reliable Digital ASIC Designs Applied to Network-Centric System Middleware Switch Processor.* PhD thesis, BTU Cottbus-Senftenberg, 2013.

[90] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[91] W. Rao, A. Orailoglu, and R. Karri. Architectural-level fault tolerant computation in nanoelectronic processors. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 533 – 539, 2-5 2005.

[92] Wenjing Rao, A. Orailoglu, and R. Karri. Fault tolerant nanoelectronic processor architectures. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 311–316 Vol. 1, Jan 2005.

[93] K. Reick, P.N. Sanda, S. Swaney, J.W. Kellington, M.J. Mack, M.S. Floyd, and D. Henderson. Fault-tolerant design of the IBM Power6 microprocessor. *Micro, IEEE*, 28(2):30 –38, march-april 2008.

[94] S.K. Reinhardt and S.S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 25 – 36, 2000.

[95] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists.* Verlag: Academic Press, 4th edition, 2009.

[96] Nelson S. Saks, M.G. Ancona, and J.A. Modolo. Generation of Interface States by Ionizing Radiation in Very Thin MOS Oxides. *Nuclear Science, IEEE Transactions on*, 33(6):1185–1190, Dec 1986.

[97] J.R. Samson, J. Ramos, A.D. George, M. Patel, and R. Some. Technology validation: NMP ST8 Dependable Multiprocessor Project. In *Aerospace Conference, 2006 IEEE*, page 14 pp., 0-0 2006.

[98] T. Sato and T. Funaki. Dependability, power, and performance trade-off on a multicore processor. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 714 –719, march 2008.

[99] C. Schluender. Device reliability challenges for modern semiconductor circuit design – a review. 2009.

[100] M. Scholzel. HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 723 –728, march 2010.

[101] Mario Scholzel. Software-based self-repair of statically scheduled superscalar data paths. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 66 –71, april 2010.

[102] Gunter Schoof, Michael Methfessel, and Rolf Kraemer. Fault-tolerant ASIC design for high system reliability. In *Smart System Integration (SSI) 2010, European Conference and Exhibition on Integration Issues of Miniaturized Systems*, March 2010.

[103] K.-D. Schubert, W. Roesner, J. M. Ludden, J. Jackson, J. Buchert, V. Paruthi, M. Behm, A. Ziv, J. Schumann, C. Meissner, J. Koesters, J. Hsu, and B. Brock. Functional verification of the IBM POWER7 microprocessor and POWER7 multiprocessor systems. *IBM Journal of Research and Development*, 55(3):10:1 –10:17, may-june 2011.

[104] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 405 –408, june 2000.

[105] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeff Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Tom Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro*, 32(2):8–19, 2012.

[106] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The case for lifetime reliability-aware microprocessors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 276 – 287, june 2004.

[107] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, June 2004.

[108] Milos Stanisavljevic, Alexandre Schmid, and Yusuf Leblebici. *Reliability of Nanoscale Circuits and Systems - Methodologies and Circuit Architectures*. Springer, 1st edition, 2011.

[109] P. Subramanyan, V. Singh, K.K. Saluja, and E. Larsson. Energy-efficient fault tolerance in chip multiprocessors using Critical Value Forwarding. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 121–130, June 2010.

[110] P. Subramanyan, V. Singh, K.K. Saluja, and E. Larsson. Multiplexed redundant execution: A technique for efficient fault-tolerance in chip multiprocessors. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1572 –1577, 8-12 2010.

[111] G.M. Swift, D.J. Padgett, and A.H. Johnston. A new class of single event hard errors. *Nuclear Science, IEEE Transactions on*, 41(6):2043–2048, Dec 1994.

[112] Synopsys. Using DesignWare ARC Processors to enhance your next SoC Design. www.synopsys.com/IP/ProcessorIP, 2011.

[113] Kohtaro Takaesu and Takeo Yoshida. Construction of a fault-tolerant voter for N-modular redundancy. *Electronics and Communications in Japan (Part II: Electronics)*, 87:62–71, December 2004.

[114] Emmanuel Touloupis, James A. Flint, Vassilios A. Chouliaras, and David D. Ward. A Fault-Tolerant Processor Core Architecture for Safety-Critical Automotive Applications. In *SAE 2005 World Congress and Exibition*, 2005.

[115] B. Turumella and M. Sharma. Assertion-based verification of a 32 thread SPARCtm CMT microprocessor. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 256 –261, june 2008.

[116] M. Vayrynen, V. Singh, and E. Larsson. Fault-tolerant average execution time optimization for general-purpose multi-processor system-on-chips. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 484 –489, 20-24 2009.

[117] J.C. Vazquez, V. Champac, A.M. Ziesemer, R. Reis, I.C. Teixeira, M.B. Santos, and J.P. Teixeira. Low-sensitivity to process variations aging sensor for automotive safety-critical applications. In *VLSI Test Symposium (VTS), 2010 28th*, pages 238 –243, 19-22 2010.

[118] X. Vera, J. Abella, J. Carretero, P. Chaparro, and A. Gonzalez. Online error detection and correction of erratic bits in register files. In *Proceedings of the 15th IEEE International On-Line Testing Symposium (IOLTS'09)*, pages 81–86, June 2009.

[119] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, April 1967.

[120] Guozhang Wang, Qiaolin Shi, Zhiguo Yu, and Zongguang Yu. The study of HW/SW co-verification on ARM-prototype system. In *Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference on*, pages 1847 –1850, oct. 2008.

[121] Kaijie Wu and R. Karri. Algorithm level recomputing with allocation diversity: a register transfer level time redundancy based concurrent error detection technique. In *Test Conference, 2001. Proceedings. International*, pages 221–229, 2001.

[122] Hyunbean Yi, Tomokazu Yoneda, Michiko Inoue, Yasuo Sato, Seiji Kajihara, and Hideo Fujiwara. Aging test strategy and adaptive test scheduling for SoC failure prediction. In *Proceedings of the 16th IEEE International On-Line Testing Symposium (IOLTS'10)*, pages 21–26, 2010.

[123] L. Zhang, J. P Zhou, J. Im, P.S. Ho, O. Aubel, C. Hennesthal, and E. Zschech. Effects of cap layer and grain structure on electromigration reliability of Cu/low-k interconnects for 45 nm technology node. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pages 581–585, May 2010.

[124] Lei Zhang, Yinhe Han, Qiang Xu, and Xiaowei Li. Defect Tolerance in Homogeneous Manycore Processors Using Core-Level Redundancy with Unified Topology. pages 891 –896, mar. 2008.

[125] Shijian Zhang and Weiwu Hu. Fetching primary and redundant instructions in turn for a fault-tolerant embedded microprocessor. In *Dependable Computing, 2008. PRDC '08. 14th IEEE Pacific Rim International Symposium on*, pages 1 –8, 15-17 2008.

[126] Hao Zhou and Jingfei Jiang. CSHFt: A Composite Fault-Tolerant Architecture and Self-Adaptable Hierarchical Fault-Tolerant Strategy for Satellite System. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2011 Tenth International Symposium on*, pages 333 –337, oct. 2011.

[127] Huiyang Zhou. A case for fault tolerance and performance enhancement using chip multi-processors. *Computer Architecture Letters*, 5(1):22–25, 2006.

[128] George W. Zobrist. *VLSI Fault Modeling and Testing Techniques*. Praeger, January 1993.