

Chapter 2

Background and Related Work

This chapter presents the background knowledge regarding different sources of the emerging reliability threats (i.e., soft errors, process variation, and aging-induced effects), the related work on soft error modeling, and their mitigation techniques. In particular, Sect. 2.1 provides the background regarding soft errors, starting with the basic transistor structure and its functionality, followed by various soft error sources and the soft error mechanism. Section 2.2 presents the basics of the NBTI-induced aging phenomena. Section 2.3 presents different variability sources and manufacturing induced process variation effects along with the process variation model explained in Sect. 2.3.1. Section 2.4 discusses the related work on soft error modeling and estimation at both the hardware and software layers. Starting from the traditional to more advanced approaches, Sect. 2.5 presents state-of-the-art soft error mitigation techniques at both hardware and software levels. As the focus of this work is on soft errors, most of the background discussed is related to soft errors. Towards the end, Sect. 2.6 summarizes the related work.

2.1 Soft Error

2.1.1 Transistor Structure

Before going into the details of how soft errors becomes an issue in the transistors, it is important to have a basic knowledge of the transistor's structure and functionality. The fundamental unit of the CMOS (Complementary Metal-Oxide-Semiconductor) microprocessor's underlying structure is the *n-type* (NMOS) and *p-type* (PMOS) transistors. The n-type transistor carries the electrons, whereas the p-type transistor carries the holes from source to drain. Figure 2.1 shows the structure of the NMOS transistor. A metal gate is attached to a thin layer of a silicon dioxide (SiO_2) which forms the interface with the silicon substrate. Channel formation is necessary for the

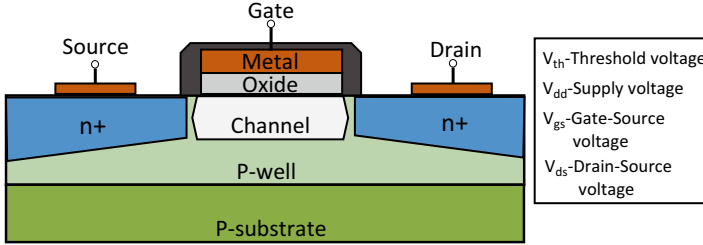


Fig. 2.1 Structure of an NMOS transistor

flow of mobile charge carriers (i.e., electrons/holes) through the transistor. A channel will be formed from source to drain and the transistor will be considered as ON, when the gate-source voltage V_{gs} becomes larger than the threshold voltage V_{th} . Otherwise, the transistor remains OFF as no channel is formed and no electrons/holes are travelling from source to drain.

2.1.2 Soft Errors in Transistors

Soft errors may be caused due to *external events* like energetic particle strikes, and/or *internal disruptive events* like noise transients at circuit, chip or system level, cross talks, and electromagnetic interference [39]. As compared to alpha particles and low energy thermal neutrons, the high-energy cosmic neutrons produce huge amount of energy (e.g., in the range of 80 MeV–1 GeV) when it strikes the nuclei of a silicon substrate in the chip [18, 135]. Figure 2.2 illustrates the soft error mechanism that can be explained in the form of the following three main phases [18].

1. **Phase-1: Ion-Track Formation:** When an energetic particle (like high-energy neutron from cosmic rays) strikes the semiconductor material (e.g., substrate of a transistor), the energy transfer results in the generation of numerous electron–hole pairs and a high carrier concentration along the ion’s path. This is called the *ion track*. The high-energy cosmic neutron produces 80 MeV–1 GeV of energy with a single strike and every 3.6 eV of energy the ion loses, produces one electron–hole pair.
2. **Phase-2: Ion Drift and Current Pulse Generation:** The produced charge is collected within a few microns at the junction. In this ion drift phase, when the ions come close to the depletion region, the electric field collects the carriers that results in a generation of a current spike (or voltage transient). This process is called “funneling.” Collection of charge near the depletion region can result in a temporary formation of a channel (even if the transistor is originally in the OFF state) and consequently leads to the flow of electrons from source to drain. This sudden current glitch may result in an instantaneous power-on of the transistor for a very short period of time (typically for tens of picoseconds).
3. **Phase-3: Ion Diffusion:** Over the period of time, the ions are diffused in the transistor for instance in the depletion region. This illustrates the transient nature of

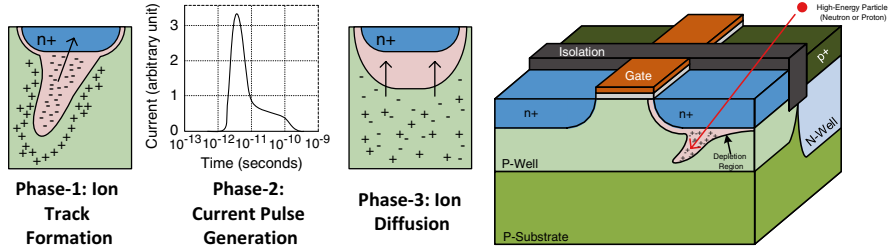


Fig. 2.2 Soft error mechanism illustrating different phases of charge generation, collection, and diffusion

soft errors. In this process, additional charge is collected until all excess carriers are captured, recombined, or diffused away.

With sufficient amount of collected charge, bit flips may result in logic devices and get latched into memory elements. These bit flips may corrupt the state of the processor (i.e., content in pipeline registers, register files, caches) and jeopardize the correct application software program execution. With the miniaturization of transistors, close spacing, and reduced critical charges, increasing trends of multi-bit upset (even from a single particle strike) have recently been reported in the literature [84, 85].

Critical Charge and Collection Charge: There are two important factors related to the soft error mechanism: (1) collection charge $Q_{\text{Collection}}$ and (2) critical charge Q_{Critical} . As transistor dimensions reduce, the critical charge also decreases along with the operating and threshold voltages. The $Q_{\text{Collection}}$ is the amount of charges (i.e., electrons/holes) collected in the conducting path of the transistor (i.e., between source and drain) to form a channel. Whereas, the Q_{Critical} is the specified number of accumulated charges which are required to form a channel between source and drain. When the $Q_{\text{Collection}} \geq Q_{\text{Critical}}$, the channel can form and the electron/holes start flowing from source to drain. Now as the Q_{Critical} reduces, a lesser number of accumulated charge is required to form a channel between the source and drain [135]. Thus, the chances for experiencing a soft error become higher, because a higher number of electron/hole pairs are generated upon a particle strike that can rapidly accumulate to form a channel in the conducting path between the source and drain. In short: lower Q_{Critical} means, fewer number of collected electrons/holes can form a strong channel. Furthermore, the Q_{Critical} becomes lower with higher temperature and further aggravates the soft error rate [117]. When an energetic particle strikes the silicon substrate, the $Q_{\text{Collection}}$ is modeled using Eq. 2.1:

$$Q_{\text{collection}} = Q_{\text{all}} \exp\left(-\frac{x_c}{L_{\text{max}}}\right) \quad (2.1)$$

Typically, the $Q_{\text{Collection}}$ is in the range of 1 to several 100 fC, and its exact amount depends upon the type of the energetic particle, its path, and the dissipated energy along the path. Likewise, from the transistor perspective there are several factors upon which the amount of $Q_{\text{Collection}}$ is dependent, i.e., size, substrate structure, location of

the strike, and device state. The transient current pulse for ion-track charge collection typically has a double exponential form with rapid rise time and gradual fall time as shown below in Eq. 2.2 [105].

$$I(t) = \frac{Q_{\text{collection}}}{\tau_\alpha - \tau_\beta} \left(e^{-\frac{t}{\tau_\alpha}} - e^{-\frac{t}{\tau_\beta}} \right) \quad (2.2)$$

The parameters τ_α and τ_β denote the collection time constant and the time constant for the ion-track formation, respectively. Both these parameters are dependent upon the process with typical values for $\tau_\alpha = 164$ ps, and $\tau_\beta = 50$ ps. The Q_{Critical} denotes the amount of the critical charge required to change the data state. Its value can be expressed as Eq. 2.3 [86].

$$Q_{\text{critical}} = \int_0^{T_F} I_D(t) dt \quad (2.3)$$

The parameter I_D denotes the time-dependent drain transient current and T_F denotes the flipping time, which is the time when both the voltages at the drain and at the gate become same [86]. The above model works well for simple circuits like DRAM storage cells. However, more complex circuits like SRAM cells experience an upset when the recovery time of the cell τ_r (time taken for the struck node voltage to return to its pre-strike value) exceeds the feedback time τ_f (time taken for the struck node voltage to become latched as incorrect data) [106]. Therefore, for the computation of Q_{Critical} , $T_F = \tau_r$ if the cell recovers and $T_F = \tau_f$ if the cell upsets. In general, $T_F = \min(\tau_r, \tau_f)$.

2.1.3 Masking Sources for Soft Errors

In combinational circuits, not all soft errors in the underlying hardware propagate to the output due to different masking effects. Masking means the ability of a logic circuit to prevent soft errors from occurring/appearing at the final output. Figure 2.3 illustrates the three major masking effects namely *Logical Masking*, *Electrical Masking*, and *Latch-Window Masking*.

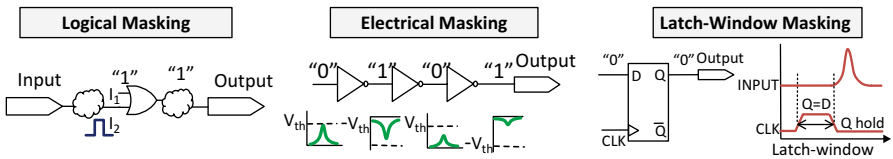


Fig. 2.3 Soft error masking effects [87]

The **logical masking** effect is defined as the capability of a logic circuit to prevent soft errors from affecting the final output. For instance, in the figure it is shown that the output of the OR gate will always be “1,” as long as I_1 maintains a logic high state (“1”). The blue pulse is caused upon a radiation event, and this will not affect the final output. In the second figure, there is an electric pulse caused by the radiation event. However, the amplitude of the pulse is not strong enough to trigger a bit flip on the input signal. The pulse will be attenuated when it passes through each gate and finally dies out. This phenomenon is called **electrical masking**. In the third figure, there is a D-latch. The value of output Q depends on the clock, if the clock is high, the value of D is set to Q, if the clock is low, Q will hold its previous value. The latch window is defined as “this” interval, during which the clock is high. If the pulse misses the window, it will not affect the final output, due to the so called **latch-window masking**.

However, logic circuits still face a lot of soft error threats in spite of the masking effects, and many state-of-the-art reliability estimation and optimization techniques are proposed at both the hardware and at the software levels (discussed in Sects. 2.4 and 2.5) to protect the logic circuit from soft errors or make sure that even if a soft error happens, the circuit can detect the errors and give the correct output.

2.2 NBTI-Induced Aging

There are different mechanisms for aging like Negative-Bias Temperature Instability (NBTI), Hot Carrier Injection, and Time-Dependent Dielectric Breakdown. NBTI-induced aging has emerged as one of the most crucial aging phenomena that happens in the PMOS transistor. An equivalent process called Positive-Bias Temperature Instability happens in NMOS.

Figure 2.4a illustrates the NBTI mechanism in a PMOS transistor which is under stress, i.e., the gate voltage is minus V_{dd} . When this voltage is applied, it creates a force at the inversion layer resulting in the breakdown of the silicon and hydrogen bond at the interface of the silicon and oxide layer (as negative stress attracts the positive hydrogen ion). The hydrogen ion is released in the oxide layer and at the broken bond a trap is created which can trap any free ions or charges, thus making the insulation imperfect. Even two neutral H atoms can combine together into an H_2 molecule, which can escape from the surface of the oxide [65]. At a higher abstraction layer, the NBTI-induced effect is manifested as an increase/shift in the threshold voltage, thereby making the transistor slower. Note, the exact phenomenon is still not precisely known and is an actively researched area in device physics [143].

The temperature plays an important role in further accelerating the NBTI-induced aging effects, i.e., increase in the threshold voltage shift. Figure 2.4b shows different curves for the threshold voltage shifts over a period of time for different operating temperatures [19]. As the temperature increases, the shift in threshold voltage aggravates and therefore increases the delay, and thus the aging effects become more prominent [151]. When estimating the change in the threshold voltages for two temperatures,

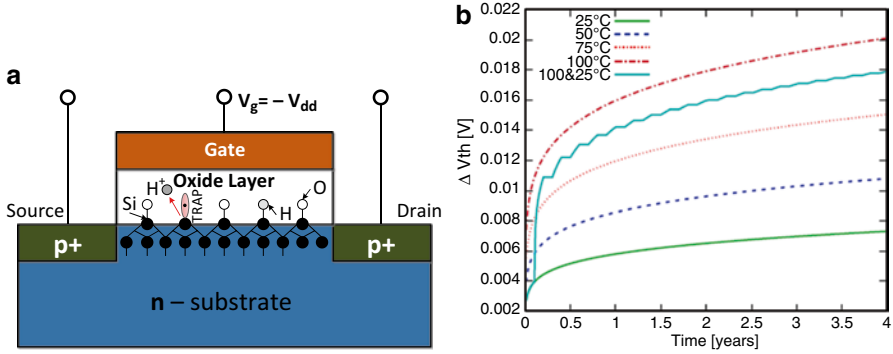


Fig. 2.4 (a) NBTI-induced aging (adapted from [137]) and (b) Impact of temperature on the NBTI-aging [19]

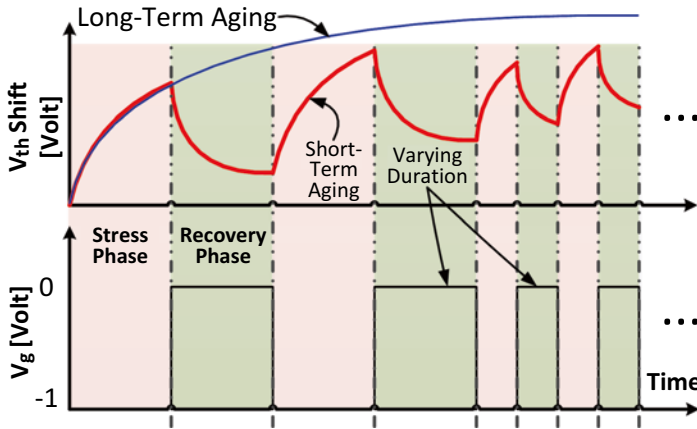


Fig. 2.5 Short-term and long-term aging [138]

i.e., 75 and 50 °C, the ΔV_{th} is approximately 50 % higher at 75 °C than 50 °C. It can be seen that the NBTI effects for two alternating temperatures, i.e., between 100 and 25 °C, are worse than that at 75 °C [151]. This shows that the shift in threshold voltage ΔV_{th} will be determined by the higher temperature.

As soon as the negative stress is removed, a recovery phase is triggered but 100 % recovery happens only in infinite time. Figure 2.5 shows an abstract view of the stress phase (causing V_{th} increase) and recovery phase (causing V_{th} decrease) along with the short-term and long-term aging effects. The stress and recovery phase can be explained using the *reaction-diffusion model* [40]. One reason for the partial recovery could be that the hydrogen ion again tries to re-bond with the silicon, but this behavior is not fully understood [142]. However, 100 % recovery is not possible as re-bonding is a random process and the same hydrogen and silicon atoms will never always perfectly re-bond. It is reported that the recovery is probably better at higher temperatures. However, higher temperatures also aggravate the threshold voltage shift in the stress phase.

If aging is not properly taken care of, then the cores' safe operating frequency (i.e., to ensure correct execution) and system clock frequency become different and may lead to timing errors. To compensate the increase in the threshold voltage V_{th} by an amount ΔV_{th} , the circuit needs to execute at a lower frequency by a factor of Δf that may violate the performance constraints, otherwise the circuit output may be faulty due to the timing errors. The industrial practice to solve this issue is via guard banding, i.e., running all the cores at the slowest frequency, which will lead to a system-wide performance loss. The aging-induced performance/delay degradation varies depending upon the stress produced due to workload and operating conditions [19]. During the first year, the aging of the core is expedited and dependent upon the core usage, whereas in the years onwards the long-term aging happens which is in times of months and years and is more dependent on the temperature [19, 138]. The device-level NBTI aging model (Eq. 2.4) employed in this work is obtained together with the *VirTherm-3D* group [151] as a part of the collaborative research effort in the DFG SPP1500 program [118]. It is based on the reaction–diffusion theory [13, 40].

$$\Delta V_{th} = 0.05 \times e^{-1500/T} \times V_{dd}^4 \times y^{1/6} \times d^{1/6} \quad (2.4)$$

ΔV_{th} is the mean threshold voltage shift in volts, T is the temperature in kelvin, V_{dd} is the supply voltage in volts, y is the age of the transistor in years, and d is the duty cycle, i.e., probability that the transistor is stressed.

Note: the aging model adopted in this work is based on the reaction–diffusion theory but another aging model based on trapping–detrapping theory [143] can also be employed because the proposed algorithms and concepts are orthogonal to the aging models. The aging values (in form of core-to-core frequency degradation) serve only as an input to the proposed cross-layer reliability modeling and optimization flow for evaluating concepts related to soft error resilience under frequency variations.

2.3 Manufacturing-Induced Process Variations and Other Variability Sources

The magnitude of the *process variations* (e.g., in the channel length/geometry and random dopant fluctuations) increases with the scaling technology trends, as it is more difficult to precisely manufacture smaller transistors with exactly the specified dimensions [25, 45, 139]. One source of variability comes from the manufacturing side which manifests in the form of core-to-core frequency variations. Figure 2.6 shows the frequency variations in the Intel's 65 nm 80-core test chip, where at 1.2 V the maximum core frequency is 7.3 GHz and the minimum is 5.7 GHz. This corresponds to 25 % frequency variation on the same chip, whereas across different chips the variation will be more [42]. The cores running at different clock frequencies may lead to timing errors. To address this issue, the major industry practice is to do guard banding by running every core with the minimum frequency on the chip, i.e.,

Fig. 2.6 Frequency variation in an 80-core processor within a single die in Intel’s 65 nm technology [42]

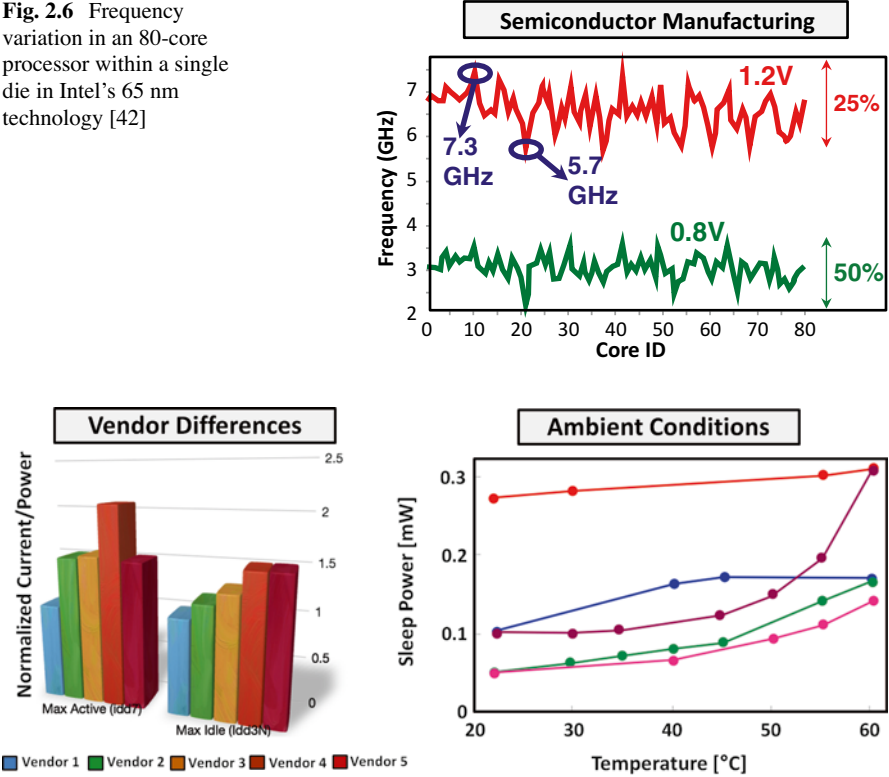


Fig. 2.7 (a) Power variation across five 512 MB DDR2-533 DRAM parts and (b) variation in sleep power (P_{sleep}) with temperature across five instances of an ARM Cortex M3 processor [26, 43]

in this case it is 5.7 GHz which is the slowest of all the cores. In synchronous system design, the core’s performance is determined by the slowest critical path, and process variations may introduce severe *design-time performance degradation*.

Another source of variability comes from different vendors. This is because different vendors use different design rules and cell libraries to fabricate the same specification, ultimately leading to variations in the leakage and dynamic power across different chips. Figure 2.7a presents the variations in the maximum active and max idle power for five different vendors fabricating the same standard, i.e., DDR2 533 DRAM chips. The standard is fixed which is the DDR2 533 but as the vendors are different there are variations in the maximum active/dynamic power and maximum idle/leakage power across different vendors.

There are also ambient conditions-dependent variabilities. This comes with the variations in the temperature, e.g., as the temperature is increased the leakage power also increases. In a single 80-core chip, different cores might have different temperatures, resulting in different leakage power and performance properties. Figure 2.7b shows the measurement of the leakage power for different temperature for different variants of five ARM cortex M3 processors. It can be seen that at the

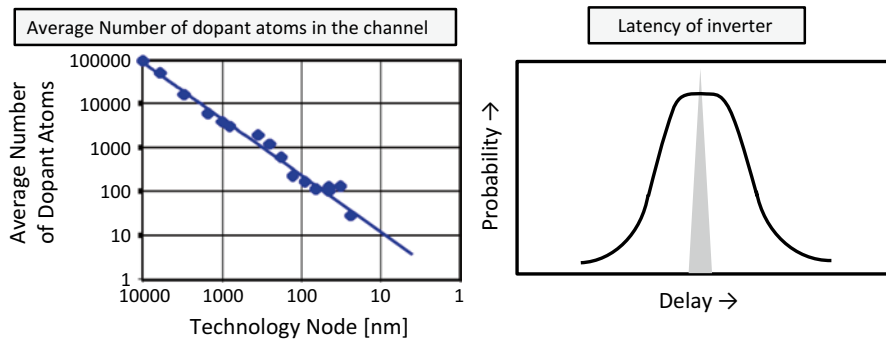


Fig. 2.8 Design-time process variation [144]

same temperature, different processors will have different leakage powers. Furthermore, a single processor shows variations in the leakage power as the temperature varies. This can be explained due to the fact that at higher temperatures, the electron mobility increases making it easier for them to escape from source to drain or from source to substrate, causing current leaks.

Figure 2.8 shows the trend of design-time variability that comes from random dopant fluctuations in the transistor. Doping is done in silicon semiconductors by implanting additional doped atoms, i.e., p+ and n+ in the transistor substrate. In the nano-scale transistors, it becomes increasingly difficult to manufacture every transistor with precisely the same number of dopant atoms, and with exactly the same dimensions, i.e., gate thickness and channel length. Earlier in one micron technology, there were more than 5000 of these dopant atoms which basically meant that minor variations of 4–5 atoms (0.001 %) had negligible effect on the electrical properties of the transistor. However, in the recent 32 nm technology, the number of dopant atoms is approximately 40 [19], and slight variations in the number of dopant atoms will significantly affect the properties of the transistor, e.g., a variation of four dopant atoms will become 10 % of the total that may result in performance degradation by approximately 10 %. The variations at the transistor level will ultimately reflect at the gate level, e.g., see the delay distribution for an inverter as shown in Fig. 2.8. In earlier technologies, the inverter was having a deterministic delay value which meant that all the inverters on the chip had exactly the same delay, e.g., 1 ps. But now due to process variations, inverters made from different transistors typically have different delay properties. For instance, some transistors may have a delay value of 1 ps while some others have 1.1 or 0.9 ps.

2.3.1 Process Variation Model

This manuscript employs the process variation model, which is proposed in [25]. It models the chip surface as a fine grid of dimensions $N_{\text{chip}} \times N_{\text{chip}}$. The process parameter value p_{ij} ($i, j \in [1; N_{\text{chip}}]$) at a grid cell (i, j) can be modeled as a Gaussian random

variable with mean μ_p and standard deviation σ_p . The correlation between the process parameters at two different grid points is given as the correlation coefficient $\rho_{ij,kl}$ that reduces with increasing distance. Based on the experimentally validated model of [44], $\rho_{ij,kl}$ is given as Eq. 2.5.

$$\rho_{ij,kl} = e^{-\alpha \sqrt{(i-k)^2 + (j-l)^2}}, \quad \forall i, j, k, l \in [1, N_{chip}] \quad (2.5)$$

The parameter α denotes the reduction rate of $\rho_{ij,kl}$. The frequency of a digital circuit can be modeled as the worst-case delay of the N_{cp} identical critical paths. According to [25, 45], the maximum frequency of core C_i ($i \in [1; N]$) in a multi-/manycore processor is modeled using Eq. 2.6, where K' is a technology-dependent constant and $S_{CP,i}$ denotes the set of N_p grid cells in C_i

$$f_i^{MAX} = K' \min_{k,l \in S_{CP,i}} \left(\frac{1}{\rho_{kl}} \right) \quad (2.6)$$

As discussed in Chap. 1, since the primary focus of this work is on the soft error related issues, in the following sections, related work for the soft error modeling and mitigation techniques will be discussed in more detail.

2.4 State-of-the-Art Soft Error Estimation Techniques

In literature, extensive research has been conducted for analyzing and modeling the soft error impacts at various granularities, i.e., at circuit-/architecture level [47, 88, 93, 94] and program level [73, 74, 76]. The standard techniques are either based upon fault injection simulations or analytical/mathematical models developed to estimate the soft error propagation across multiple gates in the combinatorial circuits. In the following, an overview of these approaches at different design abstraction layers is given.

2.4.1 Circuit-Level Techniques

In circuits, the fundamental entity in both the logic and memory parts is the NMOS and PMOS transistor which are prone to soft errors and its sensitivity grows when fabricated in scaled technology nodes [10]. The soft errors are of major concern in memories because of their large footprint in the chips. Furthermore, the soft error-induced bit flips inside memories stay unless overwritten. To handle soft errors in memories, ECC-based techniques are prominent [62, 63]. In contrast, the soft error effects in the logic part of the combinatorial circuits are relatively less frequent due to various circuit-level error masking effects (i.e., logical, electrical, and latch-window masking [87]) that prevent the errors to appear at the final output in several

cases. However, despite these error masking effects, the soft error failure rate in the combinational logic is becoming more crucial for transistors fabricated in the scaled technology nodes. This is primarily due to their reduced sizes and critical charges which makes them more prone to soft errors [21]. Moreover, achieving soft error detection and recovery in a cost-effective way is relatively difficult in combinational circuits compared to that in memories (typically protected with parity or ECC) because of the random and *transient* nature of these faults and a high degree of propagation to multiple memory elements. In general, the soft error rate measurement in circuits is in accordance with the potential error masking effects. A model is developed in [21] to measure the Soft Error Rate (SER) in the microprocessors while taking into account the effects of electrical and latch-window masking effects. At first, the SER is computed via simulating the mechanism when a particle strikes the gate till the drain of the gate. This behavior is simulated using charge to voltage pulse modes, and then the electrical model is used to check the characteristics of the voltage pulse reaching the latch input. Afterwards, for checking the pulse strength at the latch input, a pulse latching model is developed that checks the amplitude and duration to cause a soft error.

IBM developed the Soft-Error Monte Carlo Modeling program to check if the chip designs meet SER specifications [46]. Work in [47] only focuses on the electrical masking effects and has developed a mathematical model to analyze the soft error propagation across multiple gates in a combinational circuit. In [88], an analysis and modeling approach has been proposed to measure the SER while taking into account the error masking effect. Low-level HSPICE simulation is performed to obtain the electrical masking computation for each path, and logical masking computation is carried out by flipping the logic value at each input vector and each path independently. Works in [48, 49] present a reliability evaluation, where in [49], different error masking factors are separately computed for investigating the soft error tolerance of the circuit. In [48], probabilistic transfer matrices are used where each gate is represented as a matrix and for each input combination, the probability of its output value is explicitly known. However, the work presented in [48] focuses only on the logical masking effect of the circuit for given gate output probabilities without considering electrical and latch-window masking.

In [95], fault injection techniques are utilized which are based upon Monte-Carlo simulations which are time consuming because numerous experiments are performed to achieve certain accuracy. In [50], an electrical masking model is proposed in which soft error rate estimation is performed at chip level while taking into account the impact of voltage fluctuations, and it is reported that ignoring the voltage fluctuation in electrical masking can lead to inaccurate estimates of the soft error rate. Furthermore, the chances of experiencing both single and multiple bit flips are higher in the recent technologies along with the possibilities of multiple correlated bit flips. Hence it is important to account for such a correlation during soft error rate estimation, as proposed in [21]. In [51], a circuit-level technique is proposed which is based upon the error propagation probability. This technique uses a path-based analysis to check the error propagation from source to the outputs, which is a very useful and fast technique for reasonably accurate identification of the vulnerable parts of the design. In [52], a hybrid technique is proposed to compute the soft error vulnerability

of the entire microprocessor system consisting of regular and irregular structures. The techniques at the architectural and logic level are integrated for estimating the soft error vulnerability of regular (address-based structures, i.e., register file, cache) and irregular structures (i.e., logic, functional units) within a microprocessor. In general, the circuit-level techniques cannot account for the soft error masking factors at the higher system layers like architecture and software program. Therefore, a large body of research has investigated soft error estimation techniques at both architecture and software program levels.

2.4.2 Architecture-Level Techniques

The *Architectural Vulnerability Factor* (AVF) model is developed in [31] which is employed by different state-of-the-art to estimate the soft error impacts at the architectural level. The AVF of a processor component is the probability that a fault in that component will result in a visible error in its final output. It is the fraction of faults that can appear at the output in the form of user visible errors. For AVF estimation of a processor component, the *Architecturally Correct Execution* (ACE) analysis [31] is performed for all the bits in the processor component. ACE bits are the bits which are deemed necessary for architecturally correct execution, meaning that any fault in these bits will affect the correct program output when no error correction techniques are employed. All other bits are termed as un-ACE bits, because a fault in these bits will not cause a user visible erroneous program output. A bit is said to be un-ACE for a fraction of time if a fault at its value does not affect the final output of the program otherwise it is ACE. In case of storage cells, the AVF is the percentage of the time that it holds ACE bits, whereas in case of logic structures, the AVF is the fraction of time (percentage of total execution cycles) that the ACE bits or instructions are processed. It is assumed that all bits are ACE unless proven un-ACE and this may lead to overestimating the vulnerability of the target processor component. It is reported in [89] that ACE analysis overestimates the vulnerability up to 7x. Furthermore, the circuit-level masking factors such as electrical and latch-window masking in a hardware component cannot be ignored during the soft error rate analysis of a hardware component. However, the ACE analysis ignores error masking, making this approach inappropriate for *irregular hardware structures* (e.g., functional units) and only suitable for *regular structures* such as cache, register file, and reorder buffer in microprocessors [90]. It is reported that different processor components and microarchitectures have distinct AVF values, e.g., a fault in an ALU might affect the final output, whereas a fault in branch predictors might incur performance penalty but will leave no impact to the final output. The work in [73] proposed the Register Vulnerability Factor (RVF) model which considers the register bits required for architecturally correct execution. Extending the concept of AVF, the RVF models consider the fact that soft errors in the register file can be overwritten and will have no impact on the final

program output if read after being written. RVF is the probability that a soft error in registers can be propagated to other processor components (i.e., functional units, memory). While AVF concepts focus on the effect of soft error propagation, the RVF presents the probability of soft error propagation to other hardware components. The authors in [91] quantified the impact of transient faults on the Alpha 21264 microprocessor by estimating the fault masking and identifying the vulnerable portions of the processor. Enhancements of the AVF are discussed in [31, 89, 90]. The AVF is primarily used to make decisions between two reliability implementations of a processor component, but cannot be used to compare different programs at instruction and function granularity for a given architecture. A program-level reliability model is required to consider the program properties, i.e., instructions, control flow and data flow, and program-level error masking effects. A program designer also requires an error characterization from the program's perspective. Furthermore, hardware-level reliability analysis and estimation techniques [31, 91, 96] require significant development time and a long experimental duration. To address these limitations, software program-level techniques [27, 28, 75, 77, 80, 81, 92] can be used.

2.4.3 Software Program-Level Techniques

Several software program reliability estimation techniques have been proposed which are analogous to AVF. In [76], an Instruction Vulnerability Factor model is developed to assess the criticality of the instruction through fault injection experiments but this technique lacks in-depth knowledge of vulnerable bits, time-wise error probabilities, and explicit quantification of program-level masking effects. Due to the coverage issues of fault injection (especially under varying inputs), the Instruction Vulnerability Factor inherently limits accuracy of soft error analysis and estimation. Vilas et al. in [74] introduced the concept of Program Vulnerability Factor (PVF) as a microarchitecture-independent metric. PVF is fundamentally an adaptation of the AVF by shifting the Architecturally Correct Execution (ACE) analysis from microarchitecture to the program level, i.e., in a software resource (e.g., compiler-visible architectural registers). Besides PVF's inaccuracy due to ignorance of the underlying hardware properties (i.e., the layer where fault happens), PVF's consideration of only the number of ISA-visible ACE bits does not provide a comprehensive knowledge of the program reliability, and thus further limits its accuracy. The reliability of a program also depends upon the type of instructions, its data/control flow properties, and temporal effects as discussed in Chap. 3. For a same number of ACE bits, a fault in one program might cause Incorrect Output or no effect (i.e., correct output), but in another program it might cause a program failure (e.g., crash) due to the use of a different instruction. Moreover, a particle strike at a certain location in the processor may manifest as a different error compared to a strike in other parts. ACE analysis by PVF ignores different types of the manifested error that may

incur different reliability cost. PVF, however, considers all bits as ACE unless proven un-ACE which might lead to an overestimate of the program vulnerability. However, not all bits are of same vulnerability as some bits are more important for the correct execution and some might be less important. For example, faults in some bits may result in a crash and faults in some other bits may lead to data corruption, which is within the tolerable limit of the program user. A case of the tolerable limit of the program user has been demonstrated by the authors in [97] where the most significant bits are more vulnerable compared to the least significant bits. Since not all ACE bits lead to the same type of errors with the same intensity, program reliability analysis without the characterization of the manifested errors is incomplete. This instantiates the need for error characterization at the program level, given faults are injected in the underlying hardware at varying rates. Overall, the state-of-the-art approaches [31, 74, 75, 91] did not analyze the effects of changing fault rates on the program behavior when estimating the program reliability for a given system scenario. Furthermore, these reliability estimation models do not consider the knowledge of hardware-specific details, like chip footprint with processor details (e.g., area of different components, number of physical registers, fault probabilities of different processor components) for fault distribution and fault injection under different fault rates. Moreover, these techniques do not consider the time-wise error probabilities of different instructions in different components of a pipeline. Therefore, these software program-level techniques are based on abstract fault models and they will lead to over- or underestimation of the reliability.

2.4.4 Fault Injection Methodologies

Engineers typically employ fault injection to analyze and estimate the system reliability [96]. The “saboteur” or “mutant” [100] techniques are based on modifying the VHDL code. These VHDL-based techniques require precise processor models and timing details, thus requiring significant development time. Therefore, these techniques cannot be deployed in the initial design phases for the application designer. High-level simulator-based techniques typically produce the errors at the program layer in the early design phases. The technique of [101] uses a command-based injection of single-bit faults (from a fault database) in ASICs using its SystemC model. Authors in [99] propose a fault injection technique for digital signal processors. SymPLFIED is a program-level fault injection and error detection framework [98]. It enumerates transient faults in registers, memory, and computation blocks of hardware. However, it does not consider the knowledge of chip footprint in its machine model, which may lead to an inaccurate program reliability analysis. Moreover, due to its complex model evaluation it suffers from long experimental duration. The performance and analysis accuracy comparison is made against SymPLFIED in Appendix A.

2.5 State-of-the-Art Soft Error Mitigation Techniques

In order to mitigate the soft error effects, extensive research to develop reliability improvement techniques has been conducted at both the hardware level and the software level. In the following, some prominent hardware- and software-level techniques are discussed.

2.5.1 Hardware-Level Soft Error Mitigation Techniques

The hardware-level soft error mitigation techniques are tackled at the device level by adopting specialized process technology (e.g., using SOI process [55]) and materials during fabrication, at the circuit level by adopting specialized radiation hardened cells or redundant logic, and at the architecture level through redundancy in time or in space.

Device-Level Techniques: A major practice has been on exploring the possibilities of mitigating the soft error at the transistor/device level since it appears at the lower layers. The transistor-level solutions are primarily relying on the process technology, i.e., the way in which transistors are manufactured such that it becomes shielded against the radiation events like alpha particles and neutron strikes. Shielding against soft error means that the amount of collected charge $Q_{\text{collection}}$ at a transistor node once exposed to a radiation event is reduced, thus minimizing the chances of soft errors. Note, to prevent the soft error event it is important that the $Q_{\text{Collection}} < Q_{\text{Critical}}$. Adopting specialized fabrication processes and usages of some special material for fabricating soft error immune transistors is very effective, but it has a substantial overhead (in terms of, for instance, area and cost) when deployed throughout the processor. Moreover, the validation and verification costs of such approaches are considerable [19, 71, 116].

To overcome the soft errors due to the alpha particles, the semiconductor industry adopted various shielding techniques against the alpha particle-induced soft errors, e.g., by deploying thick polyimide (100 μm as stated in [53]) as an alpha particle protection layer because of their efficient thermal and electrical characteristics. With shielding, the alpha particle-originated SER is reduced to around 20 % [54]. However, such shielding solutions are typically not adopted in case of high-energy neutron-induced soft errors, because for shielding against neutron strikes, the thickness of the protection layer is required to be a minimum of approximately 10 ft in concrete, which is not feasible for almost all computing devices. As the neutron-induced SER is becoming increasingly common in the current technology, the shielding solutions do not completely eradicate the susceptibility of the device against all sources of soft error. Furthermore, deploying these techniques even for reducing the alpha particle-induced soft error is unaffordable because of the strict cost constraints.

Silicon-On-Insulator (SOI) has evolved as a promising technology for MOS/CMOS fabrications to protect against soft errors and in this way, has become superior to the conventional bulk CMOS process technology. In SOI technology, a thin-film layered insulator called buried oxide or silicon-insulator (see Fig. 2.9a) is placed in the substrate, instead of the conventional silicon substrate [55]. The transistor with SOI technology has the capability to reduce the $Q_{\text{collection}}$ upon the radiation event, because the silicon-insulator layer keeps the bulk silicon isolated, thus preventing the excess charge in the bulk silicon (induced by the radiation event) from propagating towards the source/drain, or device channel. It is reported by IBM that usage of the SOI process technology enables 5x reductions in SER for SRAM [56]. Although, the SOI technology appears to be an attractive option for reducing the SER and also in low power applications [102], nevertheless, their high manufacturing costs have made them less famous in the products that have strict design constraints. Another, well-known device-level solution is the Triple well (TW) process technique [57] which is different from the conventional CMOS process where twin-well transistor is constructed. This process technology alleviates the device sensitivity towards single event upsets. A TW device has a buried n-well layer (“deep n-well”) that separates the p-well from the p-substrate; see Fig. 2.9b. The idea of burying the n-well layer is to collect the electrons generated in the p-well region of the NMOS device upon a particle strike before they are collected at the surface of the NMOS (source–drain channel). The gathered electrons below the p-well and at the deep n-well junction do not have an impact on the device state.

Circuit-Level Techniques: At the circuit level, the soft error problem is addressed either by deploying the radiation hardened cells [66], changing the device parameters [58], or introducing redundant circuits [59]. The radiation hardened cells are in practice and are prevailing in the electronic devices deployed in the space and military missions. Changing or tuning the device parameters may help in reducing the soft error rate, e.g., increasing the supply voltage which makes the Q_{critical} higher, hence the SER becomes lower [58]. The introduction of redundant or error detection/correction circuits into the target design has been well explored in order to recover from soft errors. A prominent example is the RAZOR approach that introduces shadow flip flops in the pipeline to recover from errors. The recovery is accomplished using a global clock gating of the pipeline and an error detection through shadow flip flops that receive a delayed clock. Figure 2.10 shows (a) organization of a processor pipeline with RAZOR flip flops (FF) after each pipeline stage, and (b) the timing of the

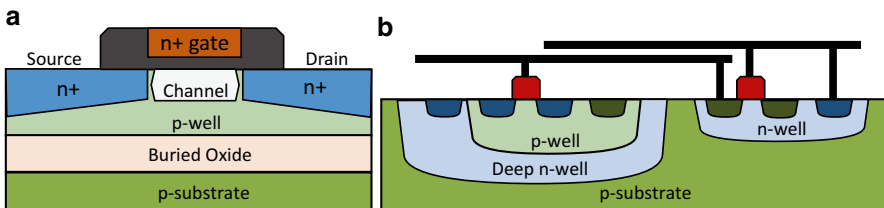


Fig. 2.9 (a) SOI MOSFET device and (b) TW NMOS FET structure

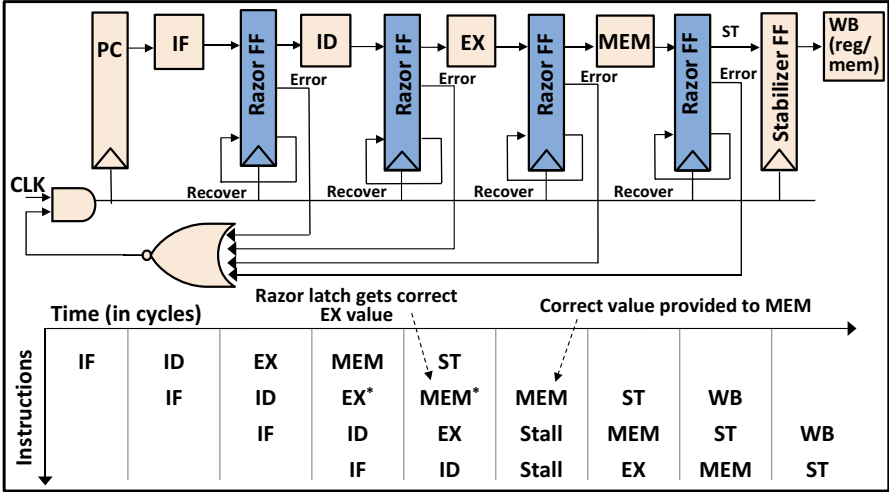


Fig. 2.10 A simple 5-stage pipelined processor with razor flip flop and error recovery [59]

overhead costs, these techniques are more practical and beneficial for the space-based applications. Furthermore, ECC and parity techniques are used to protect memories and caches. ECC-protected caches are a well-established practice in various research and industrial projects like IBM [62], AMD [63], and [35]. However, in case of register files, ECC is avoided due to high area and power overhead under frequent usage scenarios [27, 28, 73, 76], and consequently they remain vulnerable to soft errors. An approach to reduce this overhead by using the unused bits of registers is proposed in [152] but it has limited optimization potential in case of applications using full register widths and due to low soft error susceptibility of register file due to its very small footprint compared to the full pipeline. An alternate solution is employing parity-protected register files, but the error coverage is low, especially in case of multi-bit faults. Note, the contributions proposed in this manuscript are applicable to both protected and unprotected register files. Although the architecture-level solutions may not have the same precision and accuracy that the device-level solutions can offer, their efficiency is high because these solutions are independent from the underlying hardware-level details, i.e., process technology and transistor/cell structure. It is reported in [22] that when compared to the circuit-level hardening techniques, the ECC-based techniques [64] incur low area overhead. Besides the ASIC-based systems, techniques for improving the reliability of reconfigurable architectures have been proposed in [153, 154].

Redundant Multithreading Techniques: Multi-/manycore architectures facilitate soft error tolerance through excessive core availability, i.e., the cores originally reserved for performance improvements can now be exploited to improve the reliability through spatial and temporal redundancy [38, 83]. The works like fault detection via lock stepping [82], Simultaneous Redundant Threading (SRT) [67], Chip-level Redundant Threading (CRT) [36] and other works like [37] have focused

pipeline for an error that happens in the execute (EX) stage (asterisk denotes an error in the pipeline stage computation). The detailed structure of the RAZOR flip flop is shown in Fig. 2.11 that employs a shadow latch controlled by a delayed clock. Deploying additional hardware structures, however, makes the circuit-level solutions more costly due to the incurred overheads in terms of area/power and verification cost, that may become prohibitive especially in the embedded systems.

Architecture-Level Techniques: At the architectural level, the availability of different functional units with distinct structures and diverse functional and timing properties makes a wider design problem when compared to device/circuit levels. The techniques at this level are based upon the redundant executions either in *space* (using the duplicated functional units) or in *time* (using the same hardware multiple times for redundant execution and comparing the outputs). Furthermore, keeping the redundant information can also help in recovering, for instance, from the corrupted state in memory. The traditional architectural redundancy approaches are Dual Modular Redundancy (DMR), Triple Modular Redundancy (TMR), Error Correcting Code (ECC), and parity protection. The DMR approach shown in Fig. 2.12 is used for error detection where two hardware modules are used to execute the redundant copy of a code and after the execution, a comparator checks the output. In case of a mismatch, the error is detected and the rollback execution is performed for recovery.

Figure 2.12 shows TMR which is used for error detection and recovery and that employs three hardware modules for executing three copies of the code. After the execution is finished, a majority voter compares the final outputs and selects the best two out of three to determine the correct output. Besides the large area/power overhead, the voter in TMR is the single point of failure. To overcome this, triplicated voters are deployed. The increased power overhead of TMR systems may potentially increase the temperature. Increased temperatures lead to higher SER due to the reduction in the critical charge [117] and increased aging. Due to their high

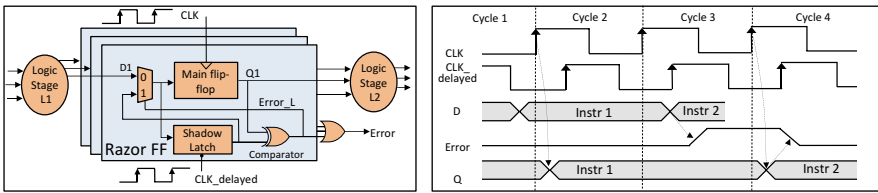


Fig. 2.11 Razor flip flop and timing diagram [59, 60]

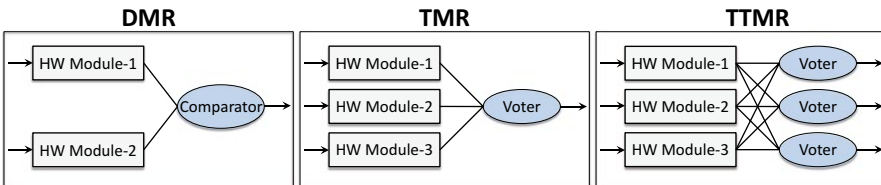


Fig. 2.12 Dual, triple, and triplicated triple modular redundancy [33, 36, 104]

on multi-/manycore based soft error mitigation where free cores are exploited to provide redundancy either at the hardware level (using redundant instructions or redundant threads) or operating system level (using redundant thread processes). Figure 2.13 presents fault detection via replicated microprocessors that are cycle-by-cycle lockstepped. This means that both the processors are synchronized with each other and have identical states at any point in time. At the same time, both processors receive same inputs and deliver the output at the same time. If an error happens in one processor, then the difference amongst the processor states will be detected and an error will be identified by the system monitor upon the output mismatch. The SRT [67] approach adapts the philosophy of the Simultaneous Multithreading (SMT) [68] approach that was originally proposed to improve the performance via executing the program codes of different applications in a simultaneous multithreaded fashion on multiple functional units inside a given processor; see Fig. 2.14. Instead of executing different application threads, SRT executes two redundant threads of the same application on multiple functional units (e.g., two adders, multipliers) inside the same core and then performs the output comparison. In contrast, the CRT approach executes redundant threads on two different processor cores; see Fig. 2.15.

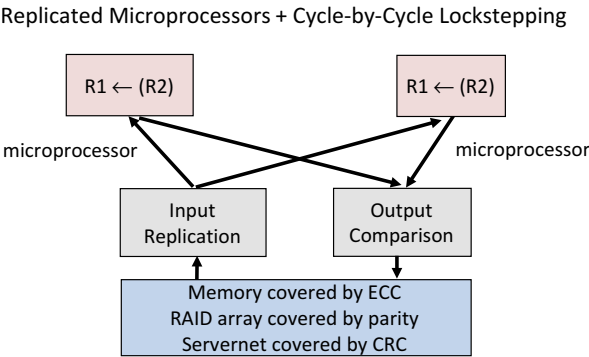


Fig. 2.13 Fault detection via lockstepping (HP Himalaya) [36]

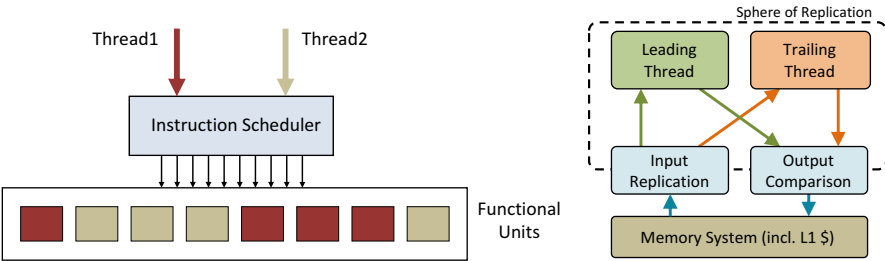
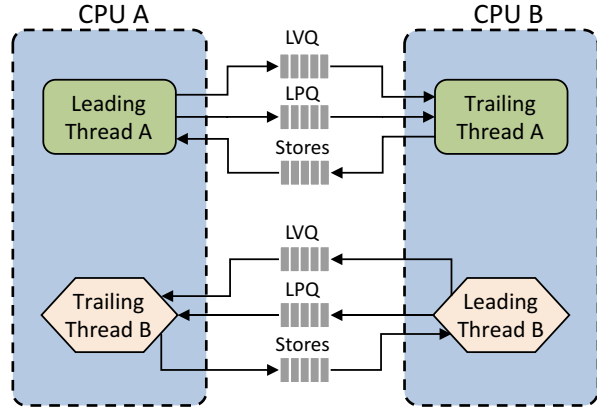


Fig. 2.14 Fault detection via simultaneous multithreading. *Left*: Scheduling different instructions on different functional units and *right*: sphere of replication with input and output replication [36, 67]

Fig. 2.15 Chip-level redundant threading [36, 67]



The SRT approach has an advantage of lower time-to-market and cost, as it exploits the existing well-established SMT architecture with little extra hardware. Moreover, it offers better performance than the complete replication. However, the challenge is the careful fetch/schedule of the redundant threads in a lockstepped fashion. The SRT combines both space- and time-wise redundancy. However, it prefers the space-wise redundancy due to its better coverage of permanent/long-duration faults. The CRT approach combines the best of SRT and lockstepping. Besides the conventional redundancy-based approaches, recent trends have evolved to explore flexible approaches, i.e., adaptive control of TMR/DMR [103], cores with heterogeneous error recovery functionalities [83], and the hardware-level checkpointing and recovery approaches [140].

The aforementioned approaches primarily target soft errors and may not address aging and process variation related problems during the soft error mitigation. For example, in CRT, two processor cores are executing redundant threads. In the presence of performance variations, one core may produce the output later than the other core. Hence, both the outputs may not synchronize for the comparison that will eventually lead to output errors. Alternatively, providing a large synchronization time may lead to performance degradation and potential deadline misses. Another limitation of these techniques is that they assume excessive area is available in the multi-/manycore system. However, in case of area-constrained embedded systems, there may be scenarios where not all applications may be supported with full DMR or TMR due to resource competition.

Summary: Within the scope of the hardware-based approaches, the soft error mitigation techniques have been explored at different abstractions in the system layers, i.e., device, circuit, and architectural level. Because of the prevailing facts that these hardware-level techniques have more area, therefore more power overhead and also high verification/validation costs, reliable hardware design and development using these techniques is both expensive and time consuming [19, 33, 71, 107, 116]. As a result, various soft error mitigation techniques at the software level have evolved. These software-level approaches are developed in various design abstraction layers which are hereby discussed in the following.

2.5.2 Software-Level Soft Error Mitigation Techniques

The classic soft error mitigation techniques (amongst many others) are N-version programming, code redundancy, control flow checking, and checkpoint recovery. The N-version programming [69] relies on implementing multiple program versions of the same specification. Depending upon the memory requirements, the number of program versions varies; however, N should not be less than 2. These N program versions are functionally identical but have diverse implementations leading to different failure characteristics such that not all versions fail in the same way under a given fault scenario. Figure 2.16 shows the working of N-version programming in a TMR model. For managing the execution, this technique demands a mechanism to synchronize the three outputs and to compare their results.

The software-based checkpoint recovery techniques [70] do not require a modification in the hardware and thereby restrict the area/power overhead. However, modification in the software is required in terms of additional implementations for the functionality. Such techniques place checkpoint instructions inside the code, typically before the critical instructions which have a high error probability. The program state is saved in reliable storage during normal execution (data checkpointing), so that in case of errors, the program can be resumed from the last checkpointed state. In [145], a compiler-assisted checkpointing scheme is proposed that inserts additional checkpointing code into user programs. An adaptive scheme is used to identify potential checkpoints in order to amortize the storage overhead of checkpointed data. The Libckpt [146] and libFT [147] checkpointing libraries provide routines to enable applications to dump critical data and/or states but require user intervention for maximum benefits. Efforts in [148] propose a reliable microkernel for application checkpointing that utilizes OS support to guarantee consistency between the current system state and process image. Besides checkpointing, a large body of work has been conducted at the software-/compiler-level that can be categorized in two major classes, (1) redundancy-based and control flow protection techniques and (2)

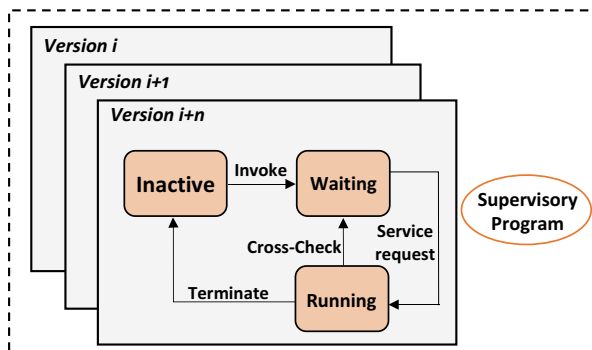


Fig. 2.16 N-version programming

Original code	Transformed Code [EDDI]
ld r12 = [GLOBAL]	ld r12 = [GLOBAL]
add r11 = r12, r13	1: ld r22 = [GLOBAL+offset]
	add r11 = r12, r13
	2: add r21 = r22,r23
	3: cmp.neq.unc p1,p0 = r11,r21
	4: cmp neq.or p1,p0 =r12,r22
	5: (p1) br faultDetected
st m[r11] = r12	st m[r11] = r12
	6: st m[r21+offset] = r22

Fig. 2.17 An example for EDDI [27]

vulnerability reduction-based techniques, as discussed below. These techniques offer new opportunities for constrained optimizations such that the cost budgets remain intact by exploiting the application characteristics.

The state-of-the-art redundancy-based techniques, such as Error Detection using Duplicated Instructions (EDDI) [27] and Software Implemented Fault Tolerance (SWIFT) [28], provide software reliability by duplicating the instructions, and inserting the comparison and checking instructions before the *store* and/or *conditional branches*. As a result, these techniques incur a significant performance overhead. An example is presented in Fig. 2.17 to explain the EDDI approach. In this example, the *load* from a global constant address is duplicated as instruction 1. In order to avoid conflicts between the original and duplicate instructions, the duplicated *load* reads its data from a different source address and stores its result into a different register. In a similar way, the *add* instruction is also duplicated as instruction 2 in order to create a redundant chain of computation. Finally, the *store* instruction is a point of synchronization, and instructions 3 and 4 compare the *store*'s operands (the address and the computed data) with their redundant copies. In case a difference is detected, instruction 5 will report an error. Otherwise, the original and its duplicate *store* instruction 6, will execute storing values to non-conflicting addresses.

As demonstrated in Fig. 2.17, this technique incurs a significant performance and memory overhead due to redundant instruction execution and shadow memory locations to store redundant data, respectively. Furthermore, performance overhead can also be attributed to the increased cache usage to hold redundant data for computation of original and duplicated instructions, generating additional memory traffic. With EDDI, although the input operands for branch instructions are verified, there is the possibility that a program's control flow gets erroneously misdirected without detection. The corruption can happen during the execution of the branch or register corruption after branch check instructions.

A well-established control flow technique is signature-based protection [34]. In order to verify that the control transfer is in the appropriate or intended basic

block, each block will be assigned a signature [34]. For that, a designated general purpose register named as GSR (General Signature Register) is employed that holds these signatures which are later used to detect faults. The GSR holds the signature value for the currently executing block. As soon as there is an entry to any block, the GSR will be *xor*'ed with a statically determined constant in order to transform the previous block's signature into the current block's signature. Once it is done the value inside the GSR can be compared with the statically assigned signature for the block to ensure that an authorized control transfer has occurred. In cases where two basic blocks have a control flow to a common block, both the blocks can jump to a common block (a control flow merge) while sharing the same signature. In such cases, using a statically determined constant to transform the GSR from the previous basic block signature to the current basic block signature might not cover control flow errors. With the statically determined constant, faults which transfer control to *or* from blocks having the same signature will remain undetected; this is undesirable. In order to avoid this, the signatures should be determined dynamically.

Figure 2.18a highlights this technique, where instruction 1 and 2 are the redundant duplicates for the *add* and *compare* instructions, respectively. Recall that, in the EDDI transformation, branches are the synchronization points. The redundant instructions from 3 to 7 are introduced in order to compare the predicate p11 to its

Original Code	(a) EDDI+ECC+CF Code	(b) EDDI+ECC+ECF Code
add r11 = r12, r13	add r11 = r12, r13	add r11 = r12, r13
cmp.lt.unc p11, p0 = r11, r12	1: add r21 = r22, r23 cmp.lt.unc p11, p0 = r11, r12 2: cmp.lt.unc p21, p0=r21, r22 3: mov r1 = 0 4: (p11) xor r1=r1, 1 5: (p21) xor r1=r1, 1 6: cmp.neq.unc p1, p0 = r1, 0 7: (p1) br faultDetected	1: add r21 = r22, r23 cmp.lt.unc p11, p0 = r11, r12 2: cmp.lt.unc p21, p0=r21, r22 3: (p21) xor RTS=sig0, sig1
(p11) br L1 L1:	(p11) br L1 L1:	(p11) br L1 4: xor RTS=sig0, sig1 L1:
st m[r11] = r12	8: xor GSR=GSR, L0_to_L1 9: cmp.neq.unc p2, p0 = GSR, sig.1 10: (p2) br faultDetected 11: cmp.neq.unc p3, p0=r11, r21 12: cmp.neq.or p3, p0=r12, r22 13: (p3) br faultDetected st m[r11] = r12	5: xor GSR=GSR, RTS 6: cmp.neq.unc p2, p0 = GSR, sig.1 7: (p2) br faultDetected 8: cmp.neq.unc p3, p0=r11, r21 9: cmp.neq.unc p3, p0=r12, r22 10: (p3) br faultDetected st m[r11] = r12

Fig. 2.18 (a) Control flow checking using software signatures [34] and (b) Enhanced control flow checking [28]

duplicate p21 and branch to error code if a fault is detected. At instruction 8, the control flow additions start that transform the GSR from the previous block signature to the signature for the currently executing block. The instructions 9 and 10 confirm if the signature is correct, in case an incorrect signature is detected, the error code is invoked. Finally, instructions 11–13 are inserted to handle the synchronization point induced by the later store instruction. This transformation will detect faulty control flow transfers between two blocks which are not unauthorized. Any such control transfer will result in an incorrect signature no matter if the erroneous transfer jumps to the middle of a basic block. These issues have led to the enhanced control flow protection approach [28]; see example in Fig. 2.18b. In this technique, for all the blocks, a dynamic equivalent of a run-time adjusting signature is used (also for the basic blocks which are not control flow merges). Each block asserts its target while using the run-time adjusted signature, and in response each target confirms the transfer by checking the GSR. Figure 2.18b demonstrates how the enhanced control flow checking works. Similar to the previous control flow checking example, in this example instructions 1 and 2 are the redundant duplicates for the add and compare instructions, respectively. The run-time signature for the target of the branch is computed via Instruction 3 by *xor*'ing the signature of the current block with the signature of the target block. As the branch is predicated, the assignment to RTS (Run-Time Signature) is also predicated using the redundant register for the predicate. Instruction 4 is the equivalent of instruction 3 for the fall through control transfer. To compute the signature of the new block, instruction 5 at the target of a control transfer *xors* RTS with the GSR. Afterwards, at instruction 6, this signature is compared with the statically assigned signature, in case of mismatch an error code is invoked with instruction 7. Just as before, instructions 8 through 9 implement the synchronization checks for the store instruction.

The SWIFT approach [28] is demonstrated in Fig. 2.19 where all the instructions are duplicated and before the *store*, the comparison instructions for fault detection are placed. In this case, it is only the *store* instructions which ultimately send data out of the *SoR* (Sphere of Replication). The system will function correctly, as long as it is ensured that *store* instructions execute only if they are “meant to” and the *store* instructions write the correct data to the correct address. This observation is used to restrict enhanced control flow checking only to blocks having the *store* instructions. In this scenario, the updates to the GSR and RTS are performed in all

Fig. 2.19 Software implemented fault tolerance (SWIFT) [28]

Original Code	Transformed Code [SWIFT]
add r1 = r2, r3	add r1 = r2, r3
mul r1 = r1, 8	1: add r1' = r2', r3' mul r1 = r1, 8
	2: mul r1' = r1', 8
	3: br faultDet, r1 != r1'
	4: br faultDet, r2 != r2'
st [r1] = r2	st [r1] = r2

blocks; however, the comparisons for signatures are restricted to blocks with *store* instructions. With this optimization, if the signature check instructions are eradicated, this will further alleviate the overhead for fault tolerance with no reduction in the reliability. Since signature comparisons are computed at the beginning of every block that contains a *store* instruction, any deviation from the valid control flow path to that point will be detected before memory and output is corrupted. This optimization has relatively lesser negative impact on the performance compared to the EDDI. Both the branch checking and enhanced control flow checking are somewhat redundant. While branch checking makes sure that branches are taken in the proper direction, the enhanced control flow checking ensures that all control transfers are made to the proper address. However, *verifying all control flow includes the notion of branching in the right direction*. Therefore, performing the control flow checking alone is adequate to detect all control flow errors.

Besides fault detection, a reliable system requires fault recovery, too. The SWIFT transformation can be seen as a DMR-like implementation that provides fault detection, but not recovery. For recovery, a TMR-like implementation SWIFTR (*Software Implemented Fault Tolerance with Recovery*) approach is presented in [71] that employs triplicated instructions and majority voting. Such full-scale redundancy solutions, however, incur significant power, area, and performance overheads. To alleviate these overheads, there are some techniques that offer enhanced control flow protection like CRAFT [28, 71] and Instruction Vulnerability Factor-based techniques [76] via duplicating only the critical instructions, i.e., instructions that have a relatively high probability to lead to a software failure/crash in case of a soft error, for instance, *load*, *store*, *jump*, *branches*, and *calls*. However, these techniques incur additional >40 % performance loss, increased register pressure due to more register usage, and excessive memory overhead because of instruction and data redundancy [28]. Furthermore, an increased number of critical instruction executions may lead to excessive rollbacks during recovery because of an increased probability of software failures and fault propagation to/from memory, when a fault occurs in the hardware of the memory pipeline stage [28]. Besides offering protection only at the instruction level, some advanced work of [77] exploits the unused bits of a register for in-register duplication, while [80] performs both the instruction and data duplication. These redundancy-based techniques incur a significant performance/memory overhead ($>2x-3x$) [27, 28, 77, 80, 92], which is typically prohibitive for embedded systems.

Besides excessive performance overhead, one of the primary issues of instruction redundancy and scheduling techniques, like [27, 28, 72, 73], is that they treat all instructions in the same way. This is because their software-level reliability estimation models (Register Vulnerability Factor¹ [73] or Program Vulnerability Factor² [74, 75]) do not distinguish between different types of errors in the software caused by the hardware-level faults during the execution

¹Register Vulnerability Factor considers the register live period as a measure for the reliability.

²Program Vulnerability Factor relates the software reliability to the bits for Architecturally Correct Execution in different programmer-visible architectural components (Register File, ALU, etc.), but hides the physical components (e.g., there are 256 physical registers, but 32 are visible to the programmer).

of different instructions that use diverse processor components in different pipeline stages. Moreover, these models are computed without considering the processor architecture. As a result, software-level reliability techniques of this kind are not very efficient. For vulnerability reduction, different compile-time approaches have evolved that seem to have promising effects on lowering the error probability of the software programs. For example, the instruction scheduling phase during the compilation can impact the instruction vulnerability by affecting the vulnerable periods of instructions and their operands in different pipeline resources. Towards this, several compile-time reliability-aware instruction scheduling approaches have been proposed [27, 76] that reorder the instruction profile of a program while incurring relatively limited performance degradation and almost no memory overhead compared to instruction redundancy techniques. The work of [77] minimizes the residency cycles of vulnerable bits inside the issue queue of superscalar processor by performing instruction scheduling at run-time. However, this technique requires architecture modification of the hardware scheduler and introduces a significant hardware overhead. In contrast, ISSE [81] reschedules a program's assembly code to minimize the operands' vulnerable periods via exploiting the slack time. The works in [78, 79] perform instruction rescheduling after the performance-optimized scheduling in order to reduce the vulnerable periods of registers. The slacks are identified after a performance-driven instruction scheduling, which already tries to minimize the slacks as much as possible to avoid pipeline stalls to improve performance. As a result, state-of-the-art instruction scheduling techniques [27] and [76] provide limited reliability improvements of 2 % and 9 %, respectively. Furthermore, the error probability is reduced by lowering the vulnerability of register file [80] or software program [28] through minimizing the register lifetime. These state-of-the-art solutions offer limited reliability improvements as they primarily improve the reliability of the register file, which typically covers a small portion of the processor layout compared to the pipeline and instruction execution unit, thus ignoring the complete processor perspective.

2.6 Summary of Related Work

In this chapter, the background related to various reliability threats, i.e., soft error, NBTI-induced aging effect, and process variation is discussed. The mechanisms of these reliability threats, their sources, and how they are modeled are presented. Since the focus of this manuscript is on soft errors, a detailed literature survey regarding the soft error estimation and mitigation techniques is presented at various levels of system abstraction, i.e., circuit level, architecture level, and program level.

Although there has been plenty of hardware-level software mitigation works at the device, circuit, and architectural layers, these techniques are not area- and power-wise cost effective as they incur extra circuitry besides their high

verification/validation costs [19, 33, 71, 107, 116]. To alleviate this overhead, various soft error mitigation techniques at the software level have evolved. The control flow checking and instruction/register value duplication result in significant performance and memory overhead, while register vulnerability reduction techniques provide limited reliability improvement. In contrast, instruction scheduling for reliability reorders the instructions of a software program without costing memory overhead and with limited/no performance overhead [73, 81]. However, these techniques ignore the complete processor perspective, as they only try to reduce the vulnerability of the register file that covers only a small portion of the processor layout compared to the complete pipeline. As a result, these techniques [73, 81] provide limited reliability improvement (2–9 %). State-of-the-art compiler-level reliability techniques have not exploited the prospective opportunities which exist at the compiler front-/middle-end optimizations that may impact the software code for improving reliability with reduced performance overhead. Furthermore, the slacks for reliability improvement are identified after a performance-driven instruction scheduling that already minimized slacks to avoid pipeline stalls to improve performance. As a result, state-of-the-art instruction scheduling techniques [73] and [81] provide limited reliability improvements of 2 % and 9 %, respectively.

In the following chapter, a comprehensive view of the novel contributions of this manuscript is presented along with details on the developed concepts, techniques, different design challenges, and motivating error analysis at the application software program level while considering the hardware-level faults.

<http://www.springer.com/978-3-319-25770-9>

Reliable Software for Unreliable Hardware

A Cross Layer Perspective

Rehman, S.; Shafique, M.; Henkel, J.

2016, XXVIII, 237 p. 121 illus., 83 illus. in color.,

Hardcover

ISBN: 978-3-319-25770-9