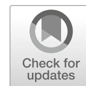
RAP Model—Enabling Cross-Layer Analysis and Optimization for System-on-Chip Resilience



Andreas Herkersdorf, Michael Engel, Michael Glaß, Jörg Henkel, Veit B. Kleeberger, Johannes M. Kühn, Peter Marwedel, Daniel Mueller-Gritschneder, Sani R. Nassif, Semeen Rehman, Wolfgang Rosenstiel, Ulf Schlichtmann, Muhammad Shafique, Jürgen Teich, Norbert Wehn, and Christian Weis

---

A. Herkersdorf (✉) · D. Mueller-Gritschneder · U. Schlichtmann
Technical University of Munich, Munich, DE, Germany
e-mail: herkersdorf@tum.de

M. Engel
Department of Computer Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
e-mail: michael.engel@ntnu.no

M. Glaß
University of Ulm, Ulm, DE, Germany

J. Henkel
Karlsruhe Institute of Technology (KIT), Karlsruhe, DE, Germany

V. B. Kleeberger
Infineon Technologies AG, Munich, DE, Germany

J. M. Kühn
Preferred Networks, Inc., Tokyo, JP, Japan

P. Marwedel
Technical University of Dortmund, Dortmund, DE, Germany

S.R. Nassif
Radyalis LLC, Austin, US, United States

S. Rehman · M. Shafique
TU Wien, Vienna, AT, Austria

W. Rosenstiel
University of Tübingen, Tübingen, DE, Germany

J. Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, DE, Germany

N. Wehn · C. Weis
University of Kaiserslautern (TUK), Kaiserslautern, DE, Germany

© The Author(s) 2021

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_1

1 Introduction/Motivation

Conquering System-on-Chip (SoC) architecture and design complexity became a major, if not the number one, challenge in integrated systems development. SoC complexity can be expressed in various ways and different dimensions: Today's single-digit nanometer feature size CMOS technologies allow for multi-billion transistor designs with millions of lines of code being executed on dozens of heterogeneous processing cores. Proving the functional correctness of such designs according to the SoC specifications is practically infeasible and can only be achieved probabilistically within tolerable margins. Further consequences of this ever-increasing hardware/software complexity are: Increasing susceptibility of application- and system-level software codes to security and safety exposures, as well as operational variability of nanometer size semiconductor devices because of environmental or manufacturing variations. The SPP1500 Dependable Embedded Systems Priority Program of the German Research Foundation (DFG) [8] focused on tackling the latter class of exposures. NBTI (negative-bias temperature instability) aging, physical electromigration damage and intermittent, radiation induced bit flips in registers (SEUs (single event upsets)) or memory cells are some manifestations of CMOS variability. The Variability Expedition program by the United States National Science Foundation (NSF) [6] is a partner program driven by the same motivation. There has been and still is a good amount of bi- and multilateral technical exchange and collaboration between the two national-level initiatives.

Divide and conquer strategies, for example, by hierarchically layering a system according to established abstraction levels, proved to be an effective approach for coping with overall system complexity in a level by level manner. Layering SoCs bottom-up with semiconductor materials and transistor devices, followed by combinatorial logic, register-transfer, micro-/macro-architecture levels, and runtime environment middleware, as well as application-level software at the top end of the hierarchy, is an established methodology used both in industry and academia. The seven layer Open Systems Interconnection (OSI) model of the International Organization for Standardization provides a reference framework for communication network protocols with defined interfaces between the layers. It is another example of conquering the complexity of the entire communication stack by layering.

Despite these merits and advantages attributed to system layering, a disadvantage of this approach cannot be overlooked. Layering fosters specialization by focusing the expertise of a researcher or developer to one specific abstraction level only (or to one layer plus certain awareness for the neighboring layers at best). Specialization and even sub-specialization within one abstraction layer became a necessity as the complexity within one layer raises already huge design challenges. However, the consequence of layering and specialization for overall system optimization is that such optimizations are typically constrained by the individual layer boundaries. Cross-layer optimization strives to pursue a more vertical approach, taking the perspectives of two or more, adjacent or non-adjacent, abstraction levels for certain system properties or qualities into account. A holistic approach (considering all abstraction levels for all system properties) is not realistic because of the overall sys-

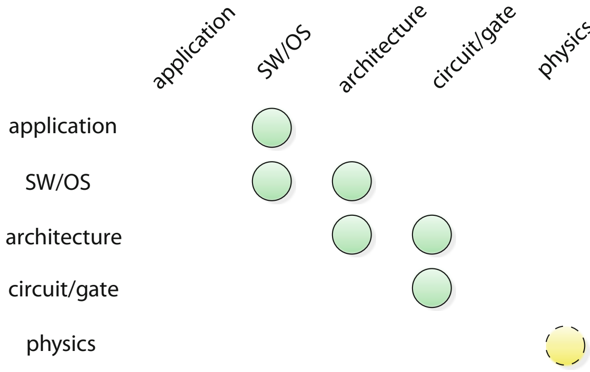


Fig. 1 RAP covers probabilistic error modeling and propagation of physics induced variabilities from circuit/logic up to application level

tem complexity. Nevertheless, for some properties, cross-layer approaches proved to be effective. Approximate computing, exploiting application-level tolerance to on-purpose circuit level inaccuracies in arithmetic operations for savings in silicon area and a lower power dissipation, is a widely adopted example of cross-layer optimization. Cross-layer approaches have also been suggested as a feasible technique to enhance reliability of complex systems [21, 26].

A prerequisite for effective cross-layer optimization is the ability to correlate the causes or events happening at one particular level with the effects or symptoms they will cause at other abstraction levels. Hierarchical system layering and specialization implies that subject matters and corresponding terminology are quite different between levels, especially when the levels of interest are several layers apart. The objective of the presented Resilience Articulation Point (RAP) model is to provision probabilistic fault abstraction and error propagation concepts for various forms of variability induced phenomena [9, 28]. Or, expressed differently, RAP aims to help annotate how variability related physical faults occurring at the semiconductor material and device levels (e.g., charge separation in the silicon substrate in response to a particle impact) can be expressed at higher abstraction levels. Thus, the impact of the low-level physical faults onto higher level fault tolerance, such as instruction vulnerability analysis of CPU core microarchitectures, or fault-aware real-time operating system middleware, can be determined without the higher level experts needing to be aware of the fault representation and error transformation at the lower levels. This cross-layer scope and property differentiates RAP from traditional digital logic fault models, such as stuck-at [18] or the conditional line flip (CLF) model [35]. These models, originally introduced for logic testing purposes, focus on the explicit fault stimulation, error propagation and observation within one and the same abstraction level. Consequently, RAP can be considered as an enabler for obtaining a cross-layer perspective in system optimization. RAP covers all SoC hardware/software abstraction levels as depicted in Fig. 1.

2 Resilience Articulation Point (RAP) Basics

In graph theory, an articulation point is a vertex that connects sub-graphs within a bi-connected graph, and whose removal would result in an increase of the number of connecting arcs within the graph. Translated into our domain of dependability challenges in SoCs, spatially and temporally correlated bit flips represent the single connecting vertex between lower layer fault origins and the upper layer error and failure models of hardware/software system abstraction (see Fig. 2).

The RAP model is based on three foundational assumptions: First, the hypothesis that every variability induced fault at the semiconductor material or device level will manifest with a certain probability as a permanent or transient single- or multi-bit signal inversion or out-of-specification delay in a signal transition. In short, we refer to such signal level misbehavior in terms of logic level or timing as a bit flip error, and model it by a probabilistic, location and time dependent error function $\mathcal{P}_{\text{bit}}(x, t)$. Second, probabilistic error functions $\mathcal{P}_L(x, t)$, which are specific to a certain abstraction layer and describe how layer characteristic data entities and compositional elements are affected by the low-level faults. For example, with what probability will a certain control interface signal on an on-chip CPU system bus, or a data word/register variable used by an application task be corrupted in response to a certain NBTI transistor aging rate. Third, there has to be a library of transformation functions $\mathcal{T}_L$ converting probabilistic error functions $\mathcal{P}_L(x_1, t)$ at abstraction level L into probabilistic error functions $\mathcal{P}_{L+i}(x_2, t + \Delta t)$ at level(s) $L + i$ $(i \geq 1)$ (see Fig. 3).

$$\mathcal{P}_{L+1}(x_2, t + \Delta t) = \mathcal{T}_L \; o \; \mathcal{P}_L(x_1, t) \tag{1}$$

Please note, although the existence of such transformation functions is a foundational assumption of the RAP model itself, the individual transformation functions

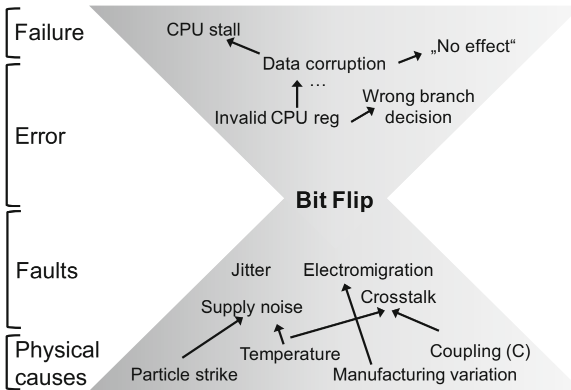


Fig. 2 Fault, error, and failure representations per abstraction levels

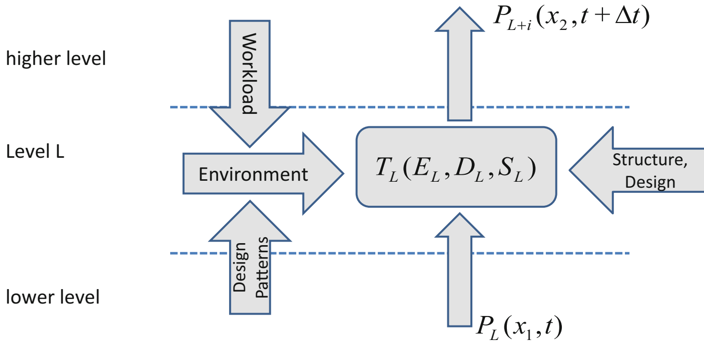


Fig. 3 Error transformation function depending on environmental, design, and system state conditions

$\mathcal{T}_L$ cannot come from or be a part of RAP. Transformation functions are dependent on a plurality of environmental, design and structure specific conditions, as well as implementation choices $(\mathcal{E}_L, \mathcal{D}_L, \mathcal{S}_L)$ within the specific abstraction layers that are only known to the respective expert designer. Note further, the location or entity x_2 affected at a higher abstraction level may not be identical to the location x_1, where the error manifested at the lower level. Depending on the type of error, the architecture of the system in use, and the characteristic of the application running, the error detection latency Δt during the root cause analysis for determining the error source at level L typically represents a challenging debugging problem [17].

3 Related Work

Related approaches to describe the reliability of integrated circuits and systems have been developed recently.

In safety-critical domains and to ensure reliable systems, standards prescribing reliability analysis approaches and MTTF (mean time to failure) calculations have been in existence for many decades (e.g., RTCA/DO-254—Design Assurance Guidance for Airborne Electronic Hardware, or the Bellcore/Telcordia Predictive Method, SR-332—Reliability Prediction Procedure for Electronic Equipment, in the telecom area [33]). These approaches, however, were not developed with automation in mind, and do not scale well to very complex systems.

The concept of reliability block diagrams (RBDs) has also been used to describe the reliability of systems [19]. In RBDs, each block models a component of the considered system. A failure rate is associated to each block. The RBD's structure describes how components interact. Components in parallel are redundant, whereas for serially connected components the failure of any one component causes the entire system to fail. However, more complex situations are difficult to model

and analyze. Such more complex situations include parametric dependencies (e.g., reliability dependent on temperature and/or voltage), redundancy schemes which can deal with certain failures, but not other (e.g., ECC which, depending on the code and number of redundant bits, can either deal with the detection and correction of single-bit failure, or detect, but not correct, multi-bit failures), or state-dependent reliability characteristics.

In 2012, RIIF (Reliability Information Interchange Format) was presented [4]. RIIF does not introduce fundamentally new reliability modeling and analysis concepts. Rather, the purpose is to provide a format for describing detailed reliability information of electronic components as well as the interaction among components. Parametric reliability information is supported. State-dependent reliability (modeled by Markov reliability models) is planned to be added. By providing a standardized format, RIIF intends to support the development of automated approaches for reliability analysis. It targets to support real-world scenarios in which complex electronic systems are constructed from legacy components, purchased IP blocks, and newly developed logic.

RIIF was developed in the context of European projects, driven primarily by the company IROC Technologies. The original concept was developed mostly within the MoRV (Modeling Reliability under Variation) project. Extensions from RIIF to RIIF2 were recently developed in collaboration with the CLERECO (Cross-Layer Early Reliability Evaluation for the Computing Continuum) project. RIIF is a machine-readable format which allows the detailed description of reliability aspect of system components. The failure modes of each component can be described, depending on parameters of the component. The interconnection of components to a system can be described. RIIF originally focused only on hardware. RIIF2 has been proposed to extend the basic concepts of RIIF to also take software considerations into account [27].

4 Fault Abstraction at Lower Levels

The RAP model proposes modeling the location and time dependent error probability $\mathcal{P}_{\text{bit}}(x, t)$ of a digital signal by an error function $\mathcal{F}$ with three, likewise, location and/or time dependent parameters: Environmental and operating conditions $\mathcal{E}$, design parameters $\mathcal{D}$, and (error) state bits $\mathcal{S}$.

$$\mathcal{P}_{\text{bit}}(x, t) = \mathcal{F}(\mathcal{E}, \mathcal{D}, \mathcal{S}) \tag{2}$$

This generic model has to be adapted to every circuit component and fault type independently. Environmental conditions $\mathcal{E}$, such as temperature and supply voltage fluctuations, heavily affect the functionality of a circuit. Device aging further influences the electrical properties, concretely the threshold voltage. Other environmental parameters include clock frequency instability and neutron flux density.

System design $\mathcal{D}$ implies multiple forms of decisions making. For example, shall arithmetic adders follow a ripple-carry or carry-look-ahead architecture (enumerative decision)? What technology node to choose (discrete decision)? How much area should one SRAM cell occupy (continuous decision)? Fixing such design parameters $\mathcal{D}$ allows the designer to make trade-offs between different decisions, which all influence the error probability of the design in one way or the other.

In order to model the dependence of the error probability on location, circuit state, and time, it is necessary to include several state variables. These state variables $\mathcal{S}$ lead to a model which is built from conditional probabilities $\mathcal{P}(b_1|b_2)$, where the error probability of the bit b_1 is dependent on the state of the bit b_2. For example, the failure probability of one SRAM cell depends on the error state of neighboring SRAM cells due to the probability of multi-bit upset (MBU) [8]. For an 8T SRAM cell it also depends on the stored value of the SRAM cell as the bit flip probability of a stored "1" is different from a stored "0."

Finally, the error function $\mathcal{F}$ takes the three parameter sets $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{S}$ and returns the corresponding bit error probability $\mathcal{P}_{\text{bit}}$. The error function $\mathcal{F}$ is unique for a specific type of fault and for a specific circuit element. An error function can either be expressed by a simple analytical formula, or may require a non-closed form representation, e.g., a timing analysis engine or a circuit simulator.

In the sequel, we show by the example of SRAM memory technology, how the design of an SRAM cell (circuit structure, supply voltage, and technology node) as well as different perturbation sources, such as radiating particle strikes, noise and supply voltage drops, will affect the data bit error probability $\mathcal{P}_{\text{bit}}$ of stored data bits.

4.1 SRAM Errors

The SRAM is well known to have high failure rates already in current technologies. We have chosen two common SRAM architectures, namely the 6-transistor (6T) and 8-transistor (8T) bit cell shown in Fig. 4. For the 6T architecture we have as design choices the number of fins for the pull-up transistors (PU), the number of fins for the

6T_(PU:PG:PD)
(a)

8T_(PU:(PG:PGR):PD)
(b)

Fig. 4 Circuit schematics for standard 6T (**a**) and 8T (**b**) SRAM bit cells

pull-down transistors (PD), and the number of fins for the access transistors (PG). The resulting architecture choice is then depicted by *6T_(PU:PG:PD)*. For the 8T architecture we have additionally two transistors for the read access (PGR). Hence, the corresponding architecture choice is named *8T_(PU:(PG:PGR):PD)*.

An SRAM cell can fail in many different ways, for example:

- **Soft Error/Single Event Upset (SEU) failure**: If the critical charge Q_{crit} is low, the susceptibility to a bit flip caused by radiation is higher.
- **Static Voltage Noise Margin (SVNM) failure**: An SRAM cell can be flipped unintentionally when the voltage noise margin is too low (stability).
- **Read delay failure**: An SRAM cell cannot be read within a specified time.
- **Write Trip Voltage (WTV) failure**: The voltage swing during a write is not high enough at the SRAM cell.

We selected these four parameters, namely Q_{crit}, SVNM, Read delay, and WTV as resilience key parameters. To quantify the influence of technology scaling (down to 7 nm) on the resilience of the two SRAM architectures we used extensive Monte-Carlo simulations and predictive technology models (PTM) [12].

4.1.1 SRAM Errors due to Particle Strikes (Q_{crit})

Bit value changes in high density SRAMs can be induced by energetic particle strikes, e.g., alpha or neutron particles [34]. The sensitivity of digital ICs to such particles is rapidly increasing with aggressive technology scaling [12], due to the correspondingly decreasing parasitic capacitances and operating voltage.

When entering the single-digit fC region for the critical charge, as in current logic and SRAM devices and illustrated in Fig. 5a, lighter particles such as alpha and proton particles become dominant (see Fig. 5b). This increases not only error rates, but also their spread, as the range of lighter particles is much longer compared to residual nucleus [10].

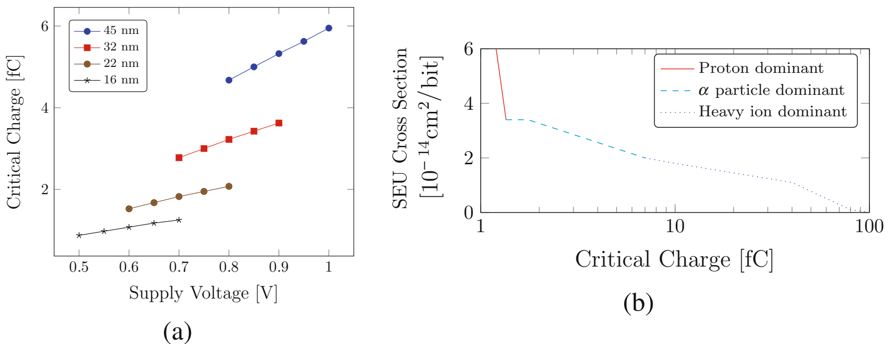


Fig. 5 Technology influence on SRAM bit flips: (**a**) Critical charge dependency on technology node and supply voltage for 6T SRAM cell, (**b**) Particle dominance based on critical charge (adapted from [10])

These technology-level faults caused by particle strikes now need to be abstracted into a bit-level fault model, so that they can be used in later system-level resilience studies. In the following this is shown for the example of neutron particle strikes. Given a particle flux of Φ, the number of neutron strikes k that hit a semiconductor area A in a time interval τ can be modeled by a Poisson distribution:

$$P(N(\tau) = k) = \exp(-\Phi \cdot A \cdot \tau) \frac{(\Phi \cdot A \cdot \tau)^k}{k!} \tag{3}$$

These neutrons are uniformly distributed over the considered area, and may only cause an error if they hit the critical area of one of the memory cells injecting a charge which is larger than the critical charge of the memory cell. The charge Q_{injected} transported by the injected current pulse from the neutron strike follows an exponential distribution with a technology dependent parameter Q_s:

$$f_Q(Q_{\text{injected}}) = \frac{1}{Q_s} \exp\left(-\frac{Q_{\text{injected}}}{Q_s}\right) \tag{4}$$

The probability that a cell flips due to this charge can then be derived as

$$P_{\text{SEU}}(Q \geq Q_{crit} | V_{cellout} = V_{DD}) = \int_{Q_{crit}}^{\infty} f_Q(Q) dQ \tag{5}$$

With increasing integration density, the probability of multi-bit upsets (MBU) also increases [16]. A comparison of the scaling trend of Q_{crit} between the 6T and 8T SRAM bit cell is shown in Fig. 6. The right-hand scale in the plots shows the 3 sigma deviation of Q_{crit} in percent to better highlight the scaling trend. The 8T-cell has a slightly improved error resilience due to an increased Q_{crit} (approximately 10% higher). However, this comes at the cost of a 25–30% area increase.

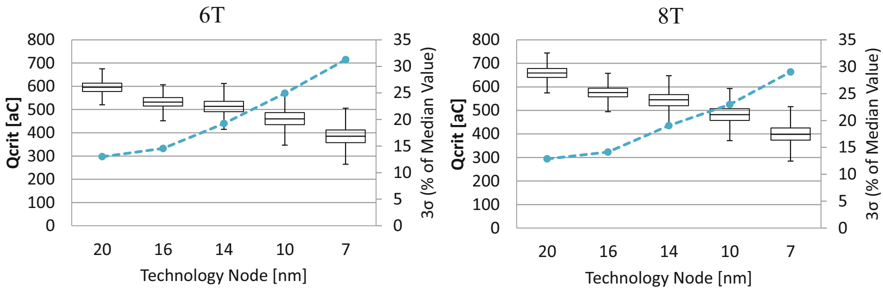


Fig. 6 Qcrit results for a 6T_(1:1:1) high density (left) and an 8T_(1:(1:1):1) (right) SRAM cell

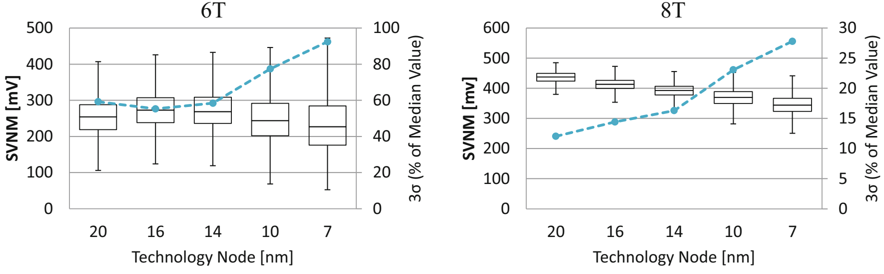


Fig. 7 SVNM results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

4.1.2 SRAM Errors due to Noise (SVNM)

The probability of an SRAM error (cell flip) due to noise is given by

$$P_{\text{noise_error}}(V_{noise} \geq V_{SVNM}) = \int_{V_{SVNM}}^{\infty} f_{V_{noise}}(V)dV \tag{6}$$

The distribution function $f_{V_{noise}}$ is not directly given as it depends largely on the detailed architecture and the environment in which the SRAM is integrated. Figure 7 plots the scaling trend for SVNM for both SRAM cell architectures. Due to its much improved SVNM the 8T_(1:(1:1):1) cell has an advantage over the 6T_(1:1:1) cell. Not only is the 8T cell approximately 22% better in SVNM than the 6T cell, but it is also much more robust in terms of 3σ variability (28% for 8T 7 nm compared to 90% for 6T 7 nm).

4.1.3 SRAM Errors Due to Read/Write Failures (Read Delay/WTV)

The probability of SRAM read errors can be expressed by the following equation:

$$P_{\text{read_error}}(t_{read} < t_{read_delay}) = \int_{0}^{t_{read_delay}} f_{t_{read}}(t)dt \tag{7}$$

In Fig. 8 the trend of the read delay for the two SRAM cell architectures is shown. Although the read delay decreases with technology scaling, which theoretically enables a higher working frequency, its relative 3σ variation can be as high as 50% at the 7 nm node. This compromises its robustness and diminishes possible increases in frequency.

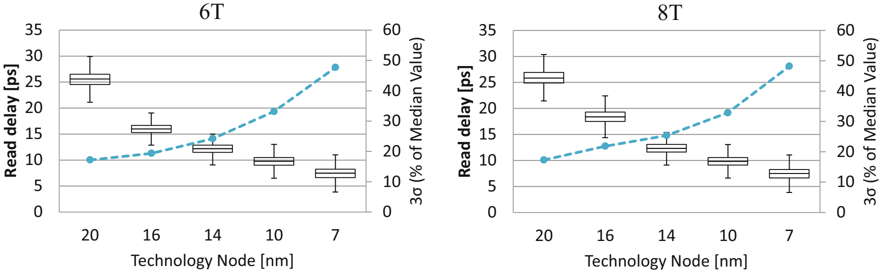


Fig. 8 Read delay results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

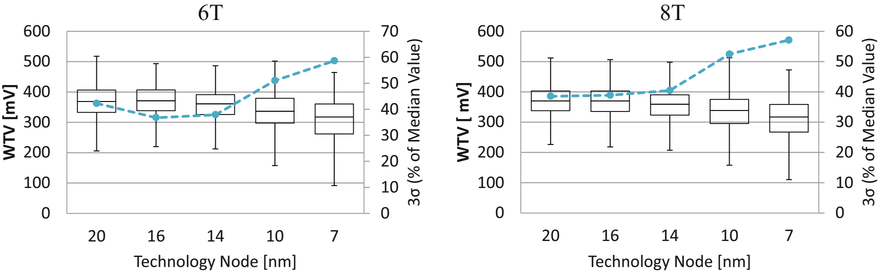


Fig. 9 WTV results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

If the actual applied voltage swing V_s is not sufficient to flip the content of a SRAM cell, then the data is not written correctly. The probability of such a write failure is given by

$$P_{\text{write_error}}(V_s < V_{swing_min}) = \int_0^{V_{swing_min}} f_{V_s}(V)dV \quad (8)$$

Similar to $f_{V_{noise}}$ both distribution functions for t_{read} and V_s depend strongly on the clock frequency, the transistor dimensions, the voltage supply, and the noise in the system. Figure 9 plots the scaling trend of WTV for 6T and 8T cells. The results for 6T and 8T cells are similar due to the similar circuit structure of 6T and 8T cells regarding write procedure.

4.1.4 SRAM Errors due to Supply Voltage Drop

Figure 10 shows the failure probability of a 65 nm SRAM array with 6T cells and 8T cells for a nominal supply voltage of 1.2 V. When the supply voltage drops below 1.2 V the failure probability increases significantly. Obviously, the behavior is different for 6T and 8T cells. The overall analysis of the resilience key parameters (Q_{crit}, SVNM, read delay, WTV, and V_{DD}) shows that the variability increases rapidly as

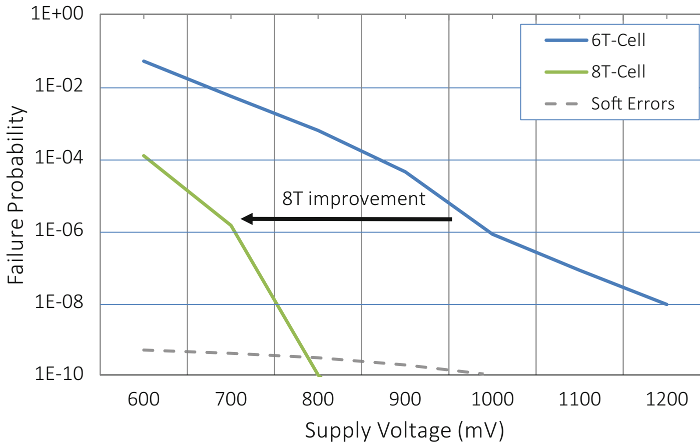


Fig. 10 Memory failure probability (65 nm technology) [1]

technology is scaled down. Investigations considering the failure probabilities of memories (SRAMs, DRAMs) in a system context are described in chapter "Design of Efficient, Dependable SoCs Based on a Cross-Layer-Reliability Approach with Emphasis on Wireless Communication as Application and DRAM Memories".

5 Architecture Level Analysis and Countermeasures

5.1 Instruction Vulnerability

Due to the wide variety in functionality and implementation of different application softwares as well as changes in the system and application workload depending on the application domain and user, a thorough yet sufficiently abstracted quantification of the dependability of individual applications is required. Even though all application software on a specific system operate on the same hardware, they use the underlying system differently, and exhibit different susceptibility to errors. While a significant number of software applications can tolerate certain errors with a relatively small impact on the quality of the output, others do not tolerate errors well. These types of errors, as well as errors leading to system crashes, have to be addressed at the most appropriate system layer in a cost-effective manner. Therefore, it is important to analyze the effects of errors propagating from the device and hardware layers to all the way up to the application layer, where they can finally affect the behavior of the system software or the output of the applications, and, therefore, become visible to the user. This implies different usage of hardware components, e.g., in the pipeline, as well as different effects of masking at the software layers while considering individual application accuracy requirements.

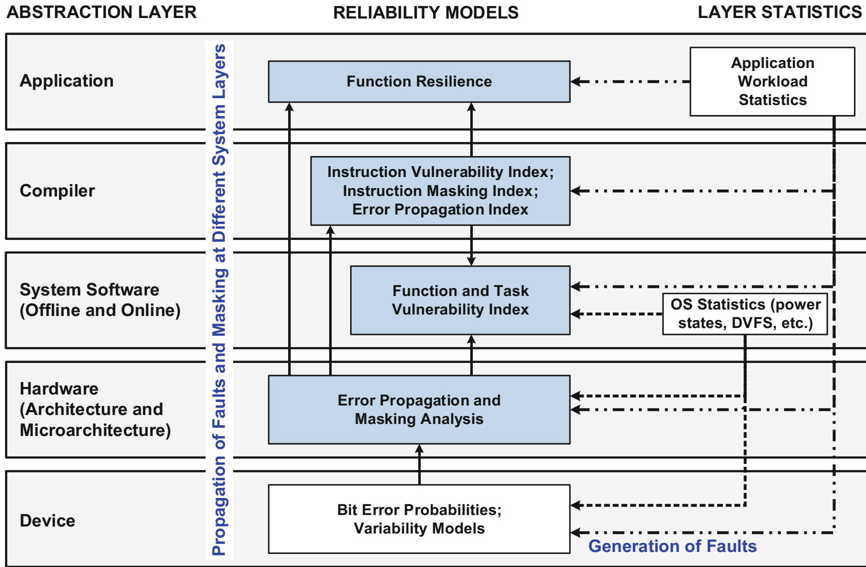


Fig. 11 Cross-layer reliability modeling and estimation: an instantiation of the RAP model from the application software's perspective

These different aspects have to be taken into account in order to accurately quantify the susceptibility of an application towards errors propagating from the lower layers.

An overview of the different models as well as their respective system layer is shown in Fig. 11 [30]. A key feature is that the software layer models consider the lower layer information while being able to provide details at the requested granularity (e.g., instruction, function, or application). To achieve that, relevant information from the lower layers has to be propagated to the upper layers for devising accurate reliability models at the software layer. As the errors originate from the device layer, a bottom-up approach is selected here. Examples for important parameters at the *hardware layer* are fault probabilities (i.e., $P_E(c)$) of different processor components ($c \in C$), which can be obtained by a gate-level analysis, as well as spatial and temporal vulnerabilities of different instructions when passing through different pipeline stages (i.e., IVI_{ic}). At the *software layer*, for instance, control and data flow information has to be considered as well as separation of critical and non-critical instructions. In addition, decisions at the OS layer (e.g., DVFS levels, mapping decisions) and application characteristics (e.g., pipeline usage, switching activity determined by data processed) can have a significant impact on the hardware. Towards that, different models have been developed on each layer and at different granularity as shown in Fig. 11. The individual models are discussed briefly in the following.

One building block for quantifying the vulnerability of an application is the *Instruction Vulnerability Index (IVI)* [22, 24]. It estimates the spatial and temporal

vulnerabilities of different types of instructions when passing through different microarchitectural components/pipeline stages $c \in C$ of a processor. Therefore, unlike state-of-the-art program level metrics (like the program vulnerability factor: PVF [32]) that only consider the program state for reliability vulnerability estimation, the IVI considers the probability that an error is observed at the output $P_E(c)$ of different processor components as well as their area A_c.

$$IVI_i = \frac{\sum_{\forall c \in C} IVI_{ic} \cdot A_c \cdot P_E(c)}{\sum_{\forall c \in C} A_c}$$

For this, the vulnerability of an instruction i in a distinct microarchitectural component c has to be estimated:

$$IVI_{ic} = \frac{v_{ic} \cdot \beta_{c(v)}}{\sum_{\forall c \in C} \beta_c}$$

The IVI_{ic} is itself based on an analysis of the vulnerable bits $\beta_{c(v)}$ representing the spatial vulnerability (in conjunction with A_c) as well as an analysis of the normalized vulnerable period v_{ic} representing the temporal vulnerability. Both capture the different residence times of instructions in the microarchitectural components (i.e., single vs. multi-cycle instructions) as well as the different usage of components (e.g., adder vs. multiplier) while combining information from the hardware and software layers for an accurate vulnerability estimation. An example for different spatial and temporal vulnerabilities is shown in Fig. 12a: Comparing an “add”- with a “load”-instruction, the “load” additionally uses the data cache/memory component (thus having a higher spatial vulnerability) and might also incur multiple stall cycles due to the access to the data cache/memory (thus having a higher temporal vulnerability).

The IVI can further be used for estimating the vulnerabilities of functions and complete application softwares. An option for a more coarse-grained model at the function granularity is the *Function Vulnerability Index (FVI)*. It models the

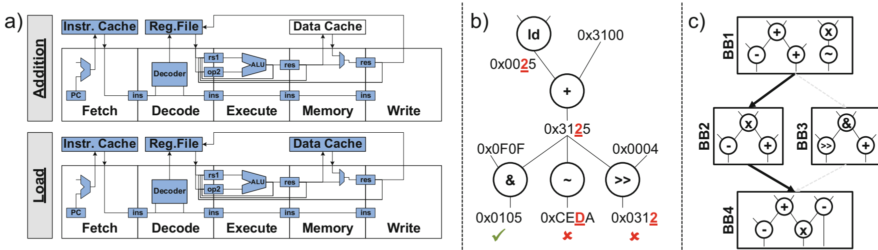


Fig. 12 (**a**) Temporal and spatial vulnerabilities of different instructions executing in a processor pipeline; (**b**) Examples for error propagation and error masking due to data flow; (**c**) Example for error masking due to control flow

vulnerability of a function as the weighted average of its susceptibility towards application failures and its susceptibility towards giving an incorrect output. In order to achieve this, critical instructions (i.e., instructions potentially causing application failures) and non-critical instructions (i.e., instructions potentially causing incorrect application outputs) are distinguished.

The quantification of the error probability provided by the IVI is complemented by capturing the masking properties of an application. The *Instruction Error Masking Index (IMI)* [31] estimates the probability that an error at instruction i is masked until the last instruction of all of its successor instruction paths. At the software layer, this is mainly determined by two factors: (a) Masking due to control flow properties, where a control flow decision might lead to an erroneous result originating from instruction i not being used (see example in Fig. 12c); (b) Masking due to data flow properties, which means that a successor instruction might mask an error originating from i due to its instruction type and/or operand values (e.g., the "and"-instruction in Fig. 12b). On the microarchitectural layer, further masking effects may occur due to an error within a microarchitectural component being blocked from propagating further when passing through different logic elements.

Although masking plays an important role, there are still significant errors which propagate to the output of a software application. To capture the effects of an error not being masked and quantify the consequences of its propagation, the *Error Propagation Index (EPI)* of an instruction can be used [31]. It quantifies the error propagation effects at the instruction granularity and provides an estimate of the extent (e.g., number of program outputs) an error at an instruction can affect the output of a software application. This is achieved by analyzing the probability that an error becomes visible at the program output (i.e., its non-masking probability) by considering all successor instructions of a given instruction i. An example of an error propagating to multiple instructions is shown in Fig. 12b.

An alternative for estimating the software dependability at the function granularity is the *Function Resilience* model [23], which provides a probabilistic measure of the function's correctness (i.e., its output quality) in the presence of faults. In order to avoid exposing the software application details (as it is the case for FVI), a black-box model is used for estimating the function resilience. It considers two basic error types: Incorrect Output of an application software (also known as Silent Data Corruption) or Application Failure (e.g., hangs, crashes, etc.). Modeling Function Resilience requires error probabilities for basic block outputs[1] and employs a Markov Chain technique; see details in [23].

As timely generation of results plays an important role, for instance, in real-time systems, it is not only important to consider the functional correctness (i.e., generating the correct output) of a software application, but also to account for the timing correctness (i.e., whether the output is provided in time or after the

[1]One potential method to obtain these error probabilities is through fault-injection experiments in the underlying hardware during the execution of these basic blocks

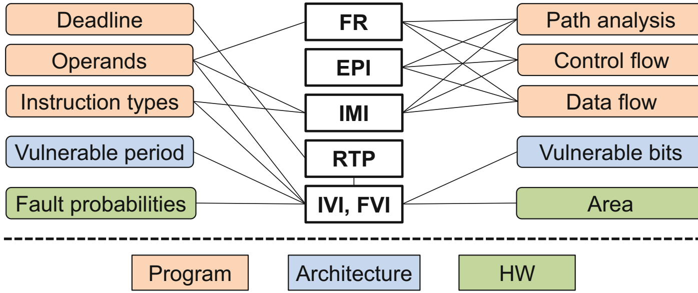


Fig. 13 Composition and focus of the different modeling approaches

deadline). This can be captured via the *Reliability-Timing Penalty (RTP)* model [25]. It is defined as the linear combination of functional reliability and timing reliability:

$$RTP = \alpha \cdot R + (1 - \alpha) \cdot miss_rate$$

where R is the reliability penalty (which can be any reliability metric at function granularity like FVI or *Function Resilience*) and $miss_rate$ represents the percentage of deadline misses for the software application. Via the parameter α ($0 \leq \alpha \leq 1$), the importance of the two components can be determined: if α is closer to 0, the timing reliability aspect is given a higher importance; when α is closer to 1, the functional reliability aspect is highlighted. The tradeoff formulated by the RTP is particularly helpful when selecting appropriate mitigation techniques for errors affecting the functional correctness, but which might have a significant time-wise overhead.

A summary of the different modeling approaches discussed above is shown in Fig. 13, where the main factors and corresponding system layers are highlighted.

5.2 Data Vulnerability Analysis and Mitigation

A number of approaches to analyze and mitigate soft errors, such as ones introduced by memory bit flips or logic errors in an ALU, rely on annotating *sections of code* as to their vulnerability to bit flips [2]. These approaches are relatively straightforward to implement, but regularly fail to capture the *context* of execution of the annotated code section. Thus, the worst-case error detection and correction overhead applies to all executions of, e.g., an annotated function, no matter what the relevance of the data processed within that function to the execution of the program (stability or quality of service effects) may be.

The SPP 1500 Program project FEHLER [29], in contrast, bases its analyses and optimizations on the notion of *data vulnerability* by performing joint *code and data flow analyses*. Here, the foremost goal is to ensure the stability of program

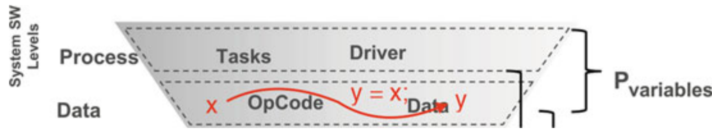


Fig. 14 Horizontal propagation of an error in the RAP model

execution while allowing a system designer to trade the resulting quality of service of a program for optimizations of different non-functional properties such as real-time adherence and energy consumption.

However, analyses on the level of single bit-flips are commonly too fine-grained for consideration in a compiler tool flow. Rather, the level of analysis provided by FEHLER allows the developer to introduce semantics of error handling above the level of single bit-flips. In the upper half of the RAP model hourglass [9], this corresponds to the "data" layer.

The seminal definition of the RAP model provides the notion of a set of bits that belong to a word of data. This allows the minimum resolution of error annotations to represent basic C data types such as `char` or `int`.[2] In addition, FEHLER allows annotations of complex data types implemented as consecutive words in memory, such as C structures or arrays.

In terms of the RAP model, data flow analyses enable the tracking of the effects of bit flips in a different dimension. The analyses capture how a hardware-induced bit error emanating in the lower half of the RAP hourglass propagates to different data objects on the same layer as an effect of arithmetic, logic, and copy operations executed by the software. As shown in Fig. 14, a bit error on the data layer can now *propagate horizontally* within the model to different memory locations. Thus, with progressing program execution, a bit flip can eventually affect more than one data object of an application.

In order to avoid software crashes in the presence of errors, affected data objects have to be classified according to the worst-case impact an error in a given object can have on a program's execution.

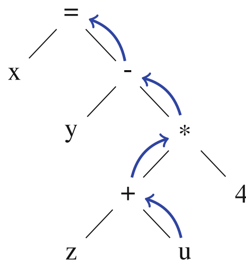
Using a bisecting approach, this results in a binary classification of the worst-case error impact of a data object on a program's execution. If an error in a data object could result in an application crash, the related piece of data is to be marked as critical to the system stability. An example for this could be a pointer variable which, in case of a bit error, might result in a processor exception when attempting to dereference that pointer. In turn, all other errors are classified as non-critical, which implies that we can ensure that a bit flip in one of these will never result in a system crash.

[2]Single bit annotations could be realized by either using C bit fields or bit banding memory areas. However, the use of bit fields is discouraged due to portability issues, whereas bit banding is not generally available on all kinds of processors and the compiler possesses no knowledge of aliasing of bit banding areas with regular memory, which would result in more complex data flow analyses.

```
unreliable int x;
reliable int y;
```

Listing 1.1 Reliability type qualifiers in FEHLER

In the FEHLER system, this classification is indicated by reliability type qualifiers, an addition to the C language that allows a programmer to indicate the worst-case effect of errors on a data object [3]. An example for possible annotations is shown in Listing 1.1. Here, the classification is implemented as extensions to the C language in the ICD-C compiler. The `reliable` type qualifier implies that the annotated data object is critical to the execution of the program, i.e., a bit flip in that variable might result in a crash in the worst case, whereas the `unreliable` type qualifier tells the compiler that the worst-case impact of a bit flip is less critical. However, in that case the error can still result in a significant reduction of a program's quality of service.




```
unreliable int u, x;
reliable int y, z;

...

x = y - (z + u) * 4;
```

Listing 1.2 Data flow analysis of possible horizontal error propagation and related AST representation

It is unrealistic to expect that a programmer is able or willing to provide annotations to each and every data object in a program. Thus, the task of analyzing the *error propagation throughout the control and data flow* and, in turn, providing reliability annotations to unannotated data objects, is left to the compiler.

An example for data propagation analysis is shown in Listing 1.2. Here, data flow information captured by the static analysis in the abstract syntax tree is used to propagate reliability type qualifiers to unannotated variables. In addition, this information is used to check the code for *invalid assignments* that would propagate permissible bit errors in `unreliable` variables to ones declared as `reliable`. Here, the `unreliable` qualifier of variable `u` propagates to the assignment to the left-hand side variable `x`. Since `x` is also declared `unreliable`, this code is valid.

```
unreliable int u, pos, tmp;
reliable int r, a[10];
u = 10;
r = u;               // invalid assignment
pos = 0;
while ( pos < r ) {  // invalid condition
  tmp = r / u;       // invalid division
  a[ pos++ ] = tmp;  // invalid memory access
}
```

Listing 1.3 Invalid assignments

Listing 1.3 gives examples for invalid propagation of data from unreliable (i.e., possibly affected by a bit flip) to reliable data objects, which are flagged as an error by the compiler.

However, there are specific data objects for which the compiler is unable to automatically derive a reliability qualifier for. Specifically, this includes input and output data, but also possibly data accessed through pointers for which typical static analyses only provide imprecise results.

The binary classification of data object vulnerability discussed above is effective when the objective is to avoid application crashes. If the quality of service, e.g., measured by the signal-to-noise ratio of a program's output, is of relevance, additional analyses are required.

FEHLER has also been applied to an approximate computing system that utilizes an ALU comprised of probabilistic adders and multipliers [7]. Here, the type qualifiers discussed before are used to indicate if a given arithmetic operation can be safely executed on the probabilistic ALU or if a precise result is required, e.g., for pointer arithmetics. The impact of different error rates on the output of an H.264 video decoder using FEHLER on probabilistic hardware is shown in Fig. 15. Here, lowering the supply voltage results in an increased error probability and, in turn, in more errors in the output, resulting in a reduced QoS as measured by the signal-to-noise ratio of the decoded video frames.

Fig. 15 Effects of different error rated on the QoS of an H.264 video decoder using FEHLER. (**a**) $V_{DD} = 1.2$ V. (**b**) $V_{DD} = 1.1$ V. (**c**) $V_{DD} = 1.0$ V. (**d**) $V_{DD} = 0.9$ V. (**e**) $V_{DD} = 0.8$ V

5.3 Dynamic Testing

Architectural countermeasures that prevent errors from surfacing or even only detect their presence come at non-neglectable costs. Whether a specific cost is acceptable or not, in turn, depends on many factors, most prominently criticality. The range of associated costs is also extensive, on one end triple modular redundancy (TMR) or similar duplication schemes such as duplication with comparison (DWC) or on the other end of the spectrum time-multiplexed methods such as online dynamic testing proposed by Gao et al. [5]. In the former examples, the costs directly correlate to the kind of assurance each technique can provide, i.e., TMR can not only continuously monitor a given component like DWC, but it can also mask any detected errors. Using TMR in the right manner, it virtually guarantees the absence of errors, but also comes at a 50% increase in both area and power consumption when compared to DWC.

Whether such cost is sensible or not depends on a complex probabilistic tradeoff with the probability of an error to occur at a specific point in time, and the criticality of an application, on the other hand, also expressed as a probabilistic term, e.g., the maximum tolerable error probability per time, often expressed as failure rate per time λ. While some applications cannot tolerate any errors such as banking transactions (or so we hope), many embedded applications have surprisingly large margins such as applications for entertainment or comfort purposes. For such applications, rather than giving absolute assurances in terms of error detection and masking (e.g., TMR or DWC), temporal limits with confidence levels are far more usable and have much higher utility for the engineering of architectural countermeasures.

Dynamic testing is a probabilistic testing scheme which can exploit such limits as its primary metric is by definition latency detection, that is the time a given dynamic testing configuration requires to detect an error with a given probability. Dynamic testing periodically samples inputs as well as associated outputs of known algorithms implemented in designated components of a SoC in a time-multiplexed fashion. Thereby obtained samples are then recomputed online on a component, the checker core, which is presumed to be more reliable. If the output sample of the device under test (DUT) does not match the recomputed sample, an error on the DUT is assumed. This testing method offers many ways to be tuned towards a specific scenario and to meet particular reliability requirements. By specifying how often a DUT is checked, how many samples per time window are being checked as well as how many such DUTs are checked using the same checker core, effort and the achievable level of assurance can be fine-tuned. Furthermore, depending on the properties of the checker core, even more ways to tailor dynamic testing towards a concrete scenario emerge.

In the presented research as demonstrated in [15], specially hardened Dynamically Reconfigurable Processors (DRPs) have been used to implement the checker functionality (See chapter ‘Increasing Reliability Using Adaptive Cross-Layer Techniques in DRPs’). DRPs are similar to FPGAs as they are reconfigurable

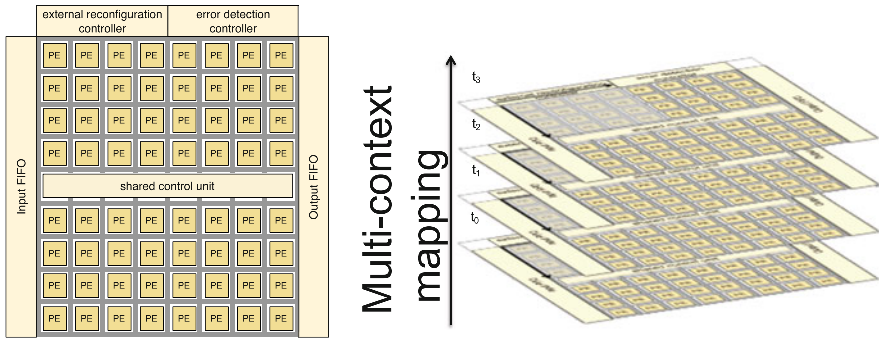


Fig. 16 General DRP structure (left) and temporal application mapping in DRPs (right)

architectures. In terms of functionality, however, they are much closer to many-core architectures, as they consist of an array of processing elements (PE) (Fig. 16 left) which operate on word granularity and possess an instruction concept combined with processor-like cycle-by-cycle internal reconfiguration. Therefore, DRPs do not only allow applications to be mapped spatially like FPGAs but also offer an extensive temporal domain to be used for better area utilization using so-called multi-context application mappings (Fig. 16 right).

For dynamic testing, this means that a DRP as a checker core is more suitable than, e.g., an embedded field programmable gate array (eFPGA) as conventional error detection ensures that the hardened DRP itself is checked regularly during non-checker operation. Furthermore, the high structural regularity also allows workloads to be shifted around on the PE array, adding additional assurances that if a DUT checks out faulty on several different PEs, the likelihood of false-positives decreases. Most importantly, however, it does not need to be dedicated to dynamic testing, but dynamic testing could be executed alongside regular applications. In turn, this, of course, also means that checker computations take longer to complete, reducing the number of samples computed per time window.

While this adaptability makes DRPs and dynamic testing an interesting match, for this combination to be useful, realistic assumptions about the error probability P are essential. If we can obtain P through, e.g., the RAP model, there are two significant advantages. Firstly, P is not constant over the lifetime of a SoC and knowledge about its distribution can help reduce testing efforts with dynamic testing. At a less error-prone time, dynamic testing allows for trade-offs such as increased time to react to errors if the error is unlikely enough to only affect a small minority of devices. Secondly, for an error with probability P to have any effect, it needs to be observable, and, thus, for all practical purposes we equate P and observation probability q which then allows us to use P to fine-tune dynamic testing to a resource minimum while meeting an upper bound for detection latency.

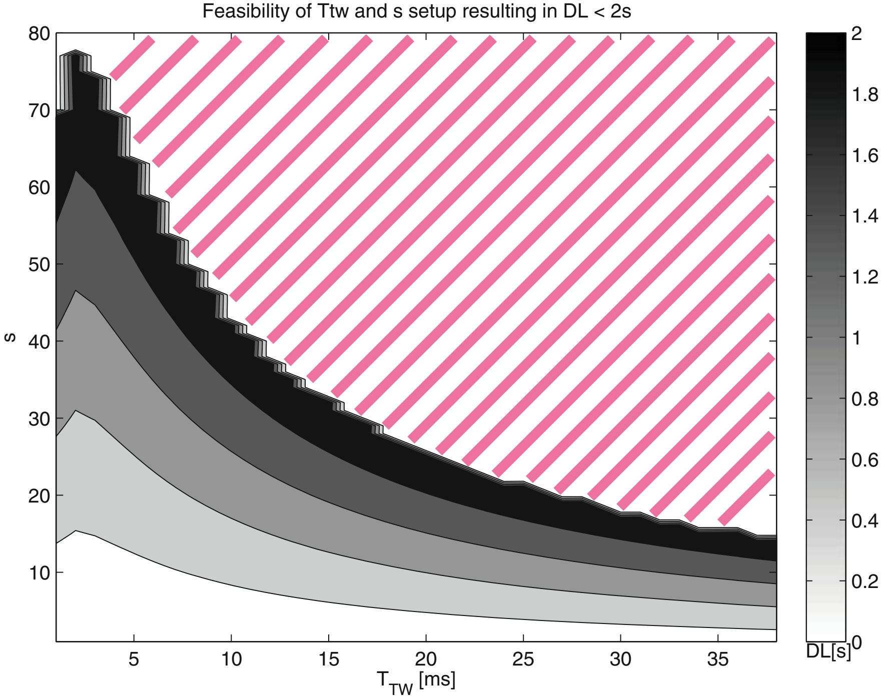


Fig. 17 Feasibility region for an error to be detected within 2 s, with $N = 4$ running at 100 MHz, an observable error probability of $P = 10^{-5}$, a reconfiguration and setup overhead of 1 ms and different scaling factors s and time windows T_{TW}

Assume a dynamic testing setup with $N = 4$ DUTs, a reconfiguration and general setup overhead of $T_{OV} = 1$ ms and time windows of $T_{TW} = \{1, \ldots, 40\,\text{ms}\}$, one round of checking requires between 8 ms and 44 ms for all DUTs. Now let s denote the scaling factor by which the temporal domain is used to map the checker functionality, e.g., $s = 3$ means using a third of the original spatial resources and, instead, prolonging the time to compute one sample by a factor of three. Consequently, a scaling factor of $s = 3$ divides the number of samples checked within one time window by three.

Now consider Fig. 17 which depicts the feasibility region by time window size T_{TW} and scaling factor s. The area which is not marked by the red dashes means that in this region, a reliability goal of a maximum detection latency DL of 2 s can be guaranteed with two-sigma confidence. However, apart from all adaptability, dynamic testing may be also waived or reduced to a minimum during times of low error probability (after early deaths in the bath tub curve). Ideally, we would only start with serious testing once the error probability is high enough to be concerned and then also only as much that the expected detection latency is within the prescribed limit. In other words, without detailed knowledge of vulnerability P, the only possibility is to guess the probabilities and add margins. If, however, P

can be estimated close enough, dynamic testing using DRPs as checker core offers a near resource optimal time and probability based technique.

Furthermore, if the characteristics of P and its development over time is understood well enough, dynamic testing could pose an alternative to DWC or even TMR for certain applications. The better P can be modeled, the smaller the margins become that have to be added to give assurances with high enough confidence. Especially for more compute intensive applications without 100% availability requirements, dynamic testing could serve as a low-cost alternative.

6 Application-Level Optimization—Autonomous Robot

Autonomous transportation systems are continuously advancing and become increasingly present in our daily lives [37]. Due to their autonomous nature, for such systems often safety and reliability are a special concern—especially when they operate together with humans in the same environment [11]. In [13], we studied the effect of soft errors in the data cache of a two-wheeled autonomous robot. The robot acts as a transportation platform for areas with narrow spacing. Due to safety reasons, the autonomous movement of the robot is limited to a predefined path. A red line on the ground, which is tracked by a camera mounted on the robot, defines the path which the robot should follow.

Since we want to study the impact of single event upsets in the data cache, the whole system memory hierarchy including accurate cache models is included in the simulation environment. We utilized in this example Instruction Set Simulation (ISS) to emulate the control SW, which consists of three main tasks: (1) the extraction of the red line from the camera frames, (2) the computation of orientation and velocity required to follow the line, and (3) evaluation of the sensor data to control the left and right motor torques to move the robot autonomously. The last task has especially hard real-time constraints because the robot must constantly be balanced. In this setup we used a fault model based on neutron particle strike induced single event upsets as shown in Sect. 4.1.1. Further, to make the fault-injection experiment feasible we used Mixture Importance Sampling to avoid simulation of irrelevant scenarios [14].

In this experiment the processor of the robot is modeled in a 45 nm technology together with a supply voltage of 0.9 V. Further, we assume a technology dependent parameter Q_s of 4.05 fC and a flux Φ of 14 Neutrons/cm^2/h (New York, Sea Level) [20, 36]. In our fault injection experiment we start with an unprotected, unhardened data cache to find the maximal resilience of the application to soft errors.

Figure 18 depicts traces of position, velocity, and orientation of the robot while it autonomously follows a line for 10 s. The injected faults lead to two types of changed system behavior:

1. strong deviations in orientation and velocity where the robot eventually loses its balance (crash sites are marked with crosses in the $x--y$ plane graph).

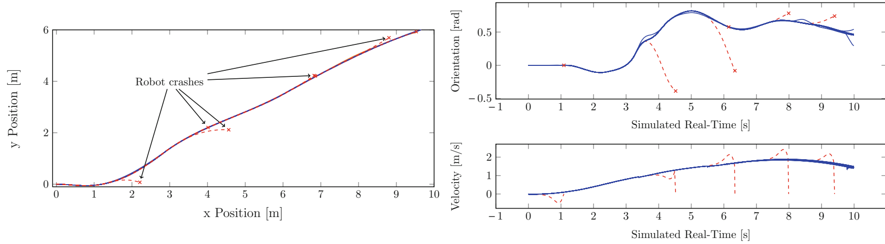


Fig. 18 Robot movement in $x - -y$ plane together with velocity and orientation angle. Dashed lines indicate crashes by CPU stalls

2. slight deviations, e.g., temporarily reduced velocity or changed orientation, where the robot still rebalances due to its feed-back control loop and still reaches its goal at the end of the line.

Further investigations showed, that for the more severe failures in (1) the simulator always reported a CPU stall. This led finally to the crash of the robot in the simulation as the balancing control was not executed any longer. Such failures are much more severe compared to (2). Still, such problems are detectable on microarchitectural level. In (2), silent data corruption (SDC) in the control algorithm happens. SDC is a severe problem for an application because it typically cannot easily be detected. Interestingly for our experiment, the algorithm shows a very high fault tolerance and often moves the robot back on its original path. This, possibly, guarantees a safe movement dependent on how narrow the robot's movement corridor is specified. The inherent error resilience of the application, thus, mitigates the SDC effect.

Based on these insights an overall cross-layer design approach for this application could look as follows: The severe crashing failures in (1) are handled by additional protection solution which detects such problems and causes a restart of the application and hence the balancing control. One typical solution to this problem is the addition of a watchdog timer to the system or a small monitoring application to key state variables of the control loop. The silent data corruption in (2) can be accepted in a certain frequency and limit according to the overall system constraints. Hence, further system design techniques and resilience actuators can be used to tune this into the required limits. This is further described in chapter 'Cross-Layer Resilience Against Soft Errors: Key Insights'.

A further use case for applying the RAP model to the cross-layer evaluation of temperature effects in MPSoC systems is presented in chapter 'Thermal Management and Communication Virtualization for Reliability Optimization in MPSoCs'.

References

1. Chang, I., Mohapatra, D., Roy, K.: A priority-based 6t/8t hybrid SRAM architecture for aggressive voltage scaling in video applications. Trans. Circuits Syst. Video Technol. **21**(2), 101–112 (2011)
2. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: an architectural framework for software recovery of hardware faults. In: Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10), pp. 497–508. ACM, New York (2010). https://doi.org/10.1145/1815961.1816026
3. Engel, M., Schmoll, F., Heinig, A., Marwedel, P.: Unreliable yet useful—reliability annotations for data in cyber-physical systems. In: Informatik 2011, Berlin, Germany, p. 334 (2011)
4. Evans, A., Nicolaidis, M., Wen, S.J.: Riif—reliability information interchange format. In: IEEE 18th International On-Line Testing Symposium (IOLTS) (2012)
5. Gao, M., Chang, H.M., Lisherness, P., Cheng, K.T.: Time-multiplexed online checking. IEEE Trans. Comput. **60**(9), 1300–1312 (2011)
6. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N.D., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., Swanson, S., Sylvester, D.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. CAD of Integr. Circuits Syst. **32**(1), 8–23 (2013). https://doi.org/10.1109/TCAD.2012.2223467
7. Heinig, A., Mooney, V.J., Schmoll, F., Marwedel, P., Palem, K., Engel, M.: Classification-based improvement of application robustness and quality of service in probabilistic computer systems. In: Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12), pp. 1–12. Springer, Berlin (2012)
8. Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M.B., Teich, J., Wehn, N., Wunderlich, H.: Design and architectures for dependable embedded systems. In: Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9–14 October 2011, pp. 69–78 (2011). https://doi.org/10.1145/2039370.2039384
9. Herkersdorf, A., et al.: Resilience articulation point (RAP): cross-layer dependability modeling for nanometer system-on-chip resilience. Microelectron. Reliab. **54**(6–7), 1066–1074 (2014). https://doi.org/10.1016/j.microrel.2013.12.012
10. Ibe, E., et al.: Spreading diversity in multi-cell neutron-induced upsets with device scaling. In: IEEE Custom Integrated Circuits Conference (CICC) (2006)
11. ISO: ISO/PAS 21448: Road vehicles—Safety of the intended functionality. International Organization for Standardization, Geneva (2019)
12. Kleeberger, V., Weis, C., Schlichtmann, U., Wehn, N.: Circuit resilience roadmap. In: Circuit Design for Reliability, pp. 121–143. Springer, Berlin (2015)
13. Kleeberger, V., et al.: A cross-layer technology-based study of how memory errors impact system resilience. IEEE Micro. **33**(4), 46–55 (2013)
14. Kleeberger, V., et al.: Technology-aware system failure analysis in the presence of soft errors by mixture importance sampling. In: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS) (2013)
15. Kühn, J.M., Schweizer, T., Peterson, D., Kuhn, T., Rosenstiel, W., et al.: Testing reliability techniques for SOCS with fault tolerant CGRA by using live FPGA fault injection. In: 2013 International Conference on Field-Programmable Technology (FPT), pp. 462–465. IEEE, New York (2013)
16. Lee, S., Baeg, S., Reviriego, P.: Memory reliability model for accumulated and clustered soft errors. IEEE Trans. Nucl. Sci. **58**(5), 2483–2492 (2011)
17. Lin, D., Hong, T., Li, Y., Fallah, F., Gardner, D.S., Hakim, N., Mitra, S.: Overcoming post-silicon validation challenges through quick error detection (QED). In: Design, Automation and Test in Europe (DATE 13), Grenoble, France, 18–22 March 2013, pp. 320–325 (2013). https://doi.org/10.7873/DATE.2013.077

18. Millman, S.D., McCluskey, E.J.: Detecting bridging faults with stuck-at test sets. In: Proceedings International Test Conference 1988, Washington, D.C., USA, September 1988, pp. 773–783 (1988). https://doi.org/10.1109/TEST.1988.207864
19. Modarres, M., Kaminskiy, M., Krivtsov, V.: Reliability Engineering and Risk Analysis: A Practical Guide. CRC Press, New York (1999)
20. Mukherjee, S.: Architecture design for soft errors. Morgan Kaufmann, Burlington (2011)
21. Quinn, H.M., De Hon, A., Carter, N.: CCC visioning study: system-level cross-layer cooperation to achieve predictable systems from unpredictable components. Tech. rep., Los Alamos National Laboratory (LANL) (2011)
22. Rehman, S., Kriebel, F., Shafique, M., Henkel, J.: Reliability-driven software transformations for unreliable hardware. IEEE Trans. CAD Integr. Circuits Syst. **33**(11), 1597–1610 (2014). https://doi.org/10.1109/TCAD.2014.2341894
23. Rehman, S., Shafique, M., Aceituno, P.V., Kriebel, F., Chen, J., Henkel, J.: Leveraging variable function resilience for selective software reliability on unreliable hardware. In: Design, Automation and Test in Europe (DATE 13), Grenoble, France, 18–22 March 2013, pp. 1759–1764 (2013). https://doi.org/10.7873/DATE.2013.354
24. Rehman, S., Shafique, M., Kriebel, F., Henkel, J.: Reliable software for unreliable hardware: embedded code generation aiming at reliability. In: Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9–14 October 2011. pp. 237–246 (2011). https://doi.org/10.1145/2039370.2039408
25. Rehman, S., Toma, A., Kriebel, F., Shafique, M., Chen, J., Henkel, J.: Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In: 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013), Philadelphia, PA, USA, 9–11 April 2013, pp. 273–282 (2013). https://doi.org/10.1109/RTAS.2013.6531099
26. Robinson, W., Alles, M., Bapty, T., Bhuva, B., Black, J., Bonds, A., Massengill, L., Neema, S., Schrimpf, R., Scott, J.: Soft error considerations for multicore microprocessor design. In: IEEE International Conference on Integrated Circuit Design and Technology, pp. 1–4. IEEE, New York (2007)
27. Savino, A., Di Carlo, S., Vallero, G., Politano, G., Gizopolous, D., Evans, A.: RIIF-2—toward the next generation reliability information interchange format. In: IEEE 22nd International On-Line Testing Symposium (IOLTS) (2016)
28. Schlichtmann, U., et al.: Connecting different worlds—technology abstraction for reliability-aware design and test. In: Design, Automation & Test in Europe Conference & Exhibition (DATE 2014), Dresden, Germany, 24–28 March 2014, pp. 1–8 (2014). https://doi.org/10.7873/DATE.2014.265
29. Schmoll, F., Heinig, A., Marwedel, P., Engel, M.: Improving the fault resilience of an H.264 decoder using static analysis methods. ACM Trans. Embed. Comput. Syst. **13**(1s), 31:1–31:27 (2013). https://doi.org/10.1145/2536747.2536753
30. Shafique, M., Axer, P., Borchert, C., Chen, J., Chen, K., Döbel, B., Ernst, R., Härtig, H., Heinig, A., Kapitza, R., Kriebel, F., Lohmann, D., Marwedel, P., Rehman, S., Schmoll, F., Spinczyk, O.: Multi-layer software reliability for unreliable hardware. it—Inf. Technol. **57**(3), 170–180 (2015). https://doi.org/10.1515/itit-2014-1081
31. Shafique, M., Rehman, S., Aceituno, P.V., Henkel, J.: Exploiting program-level masking and error propagation for constrained reliability optimization. In: The 50th Annual Design Automation Conference 2013 (DAC '13), Austin, TX, USA, May 29–June 07 2013, pp. 17:1–17:9 (2013). https://doi.org/10.1145/2463209.2488755
32. Sridharan, V., Kaeli, D.R.: Eliminating microarchitectural dependency from architectural vulnerability. In: 15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14–18 February 2009, Raleigh, North Carolina, USA, pp. 117–128 (2009). https://doi.org/10.1109/HPCA.2009.4798243
33. TelCordia Technologies: Reliability Prediction Procedure for Electronic Equipment, SR-332 (2016)

34. Wirth, G., Vieira, M., Neto, E., Kastensmidt, F.: Generation and propagation of single event transients in CMOS circuits. In: IEEE Design and Diagnostics of Electronic Circuits and systems (2006)
35. Wunderlich, H.J., Holst, S.: Generalized fault modeling for logic diagnosis. In: Models in Hardware Testing—Lecture Notes of the Forum in Honor of Christian Landrault. Springer, Berlin (2010)
36. Zhang, M., Shanbhag, N.: Soft-error-rate-analysis (sera) methodology. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(10), 2140–2155 (2006)
37. Ziegler, J., et al.: Making Bertha drive—an autonomous journey on a historic route. IEEE Intell. Transp. Syst. Mag. **6**(2), 8–20 (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.