Jörg Henkel
Nikil Dutt *Editors*

Dependable Embedded Systems

OPEN ACCESS

Springer

Embedded Systems

This Series addresses current and future challenges pertaining to embedded hardware, software, specifications and techniques. Titles in the Series cover a focused set of embedded topics relating to traditional computing devices as well as high-tech appliances used in newer, personal devices, and related topics. The material will vary by topic but in general most volumes will include fundamental material (when appropriate), methods, designs and techniques.

More information about this series at http://www.springer.com/series/8563

Jörg Henkel • Nikil Dutt
Editors

Dependable Embedded Systems

Springer

Editors
Jörg Henkel
Karlsruhe Institute of Technology
Karlsruhe, Baden-Württemberg,
Germany

Nikil Dutt
Computer Science
University of California, Irvine
Irvine, CA, USA

ISSN 2193-0155 ISSN 2193-0163 (electronic)
Embedded Systems
ISBN 978-3-030-52016-8 ISBN 978-3-030-52017-5 (eBook)
https://doi.org/10.1007/978-3-030-52017-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Wolfgang,
our inspiring colleague, co-initiator of the SPP 1500 program and a good friend. We will truly miss him.

Prof. Dr. rer. nat. Wolfgang Rosenstiel
5.10.1954–19.08.2020

Preface

Dependability has become a major issue since Moore's law had hit its limits. While Moore's law has been the pacemaker for the microelectronics age for about four decades, the exponential growth has led to the microelectronic revolution that has changed our lives in multifarious ways starting from the PC through the internet and embedded applications like safety in automotive to today's personal communication/entertainment devices. The positive side effects of this exponential growth were:

(a) Decreased Costs: This refers to the costs per transistor that decreased exponentially as complexity (i.e., number of transistors per chip) increased. In other words, for the same costs, the customer received far more functionality when migrating from one technology node to the next one.
(b) Increased Performance: Since transistors shrank, the effective capacitances shrank, too. Hence, signal delays decreased and allowed for higher clocking, i.e., the clock frequency could be raised and significant performance gains could be achieved.
(c) Decreased Power Consumption: Since smaller transistors have lower effective switching capacitances, the power consumption per transistor and the overall power consumption per chip went significantly down. This opened the opportunity for new application areas like mobile devices, etc.

In summary, Moore's law had provided a win–win situation for four decades in virtually all relevant design constraints (i.e., cost, power consumption, performance, and chip area). However, as Gordon E. Moore had already stated in a talk at ISSCC 2003: "No exponential is forever ... but we can delay 'forever'...," he indicated that the exponential growth cannot be sustained forever but that it may be possible to delay the point when scalability finally comes to an end.

However, systems in the nano-CMOS era are inherently undependable when further advancing from one technology node to the next.

In particular, we can identify the following challenging problems which negatively impact the dependability of future systems. If not addressed properly, the dependability of systems will significantly decrease.

The effects can be divided into two major groups: The first group comprises those effects that stem from fabrication/design time issues, whereas the second group stems from operation/run-time execution.

Fabrication and Design-Time Effects

Yield and Process Variations

Yield defines the number of flaw-free circuits in relation to all fabricated circuits. A high yield is so far considered vital for an economic production line. Unfortunately, the yield will dramatically decrease because feature sizes reach a point where the process of manufacturing underlies statistical variances. Future switching devices may be fabricated through "growing" or "self-assembly." All known research suggest that these processes cannot be controlled entirely, leading to fabrication flaws, i.e., circuits with faulty devices. As per the definition of yield, it will at a not-that-distant point in time go to zero, i.e., no circuit can be produced without at least a single faulty switching device. The traditional way of sorting out faulty circuits will not work any longer! Rather, faults will be inherent. On the other hand, fabricated circuits (although functionally correct) will continue to exhibit increasing levels of "process variability": i.e., a high degree of variability in the observed performance, power consumption, and reliability parameters both across manufactured parts and across use of these parts over time in the field. The traditional "guardbanding" approach of overdesigning circuits with a generous margin to hide these process variations will no longer be economically viable nor will fit into a traditional design flow that assumes a rigid specification of operational constraints for the performance, power, and reliability of manufactured circuits. Newer design techniques and metholodologies will therefore need to address explicitly the effects of process variation, rather than assuming these are hidden through traditional overdesigned guardbanding margins.

Complexity

In about 10 years from now, the complexity of systems integrated into one single die will amount to basic switching devices. The steadily increasing integration complexity is efficiently exploited by the current trend towards many-core network-on-chip architectures. These architectures introduce hardware and software complexities, which previously were found on entire printed circuit boards and systems down to a single chip and provide significant performance and power advantages in comparison with single cores. A large number of processing and communication

elements require new programming and synchronization models. It leads to a paradigm shift away from the assumption of zero design errors.

Operation and Run-Time Effects

Aging Effects

Transistors in the nano-CMOS era are far more susceptible to environmental changes like heat, as an example. It causes an irreversible altering of the physical (and probably chemical) properties which, itself, lead to malfunctions and performance variability over time. Though effects like electromigration in current CMOS circuits are well known, they typically do not pose a problem since the individual switching device's lifetime is far higher than the product life cycle. In future technologies, however, individual switching devices will fail (i.e., age) earlier than the life cycle of the system (i.e., product) they are part of. Another emergent altering effect is the increasing susceptibility to performance variability resulting in changing critical paths over time. This, for instance, prevents a static determination of the chip performance during manufacturing tests.

Thermal Effects

Thermal effects will have an increasing impact on the correct functionality. Various degradation effects are accelerated by thermal stress like very high temperature and thermal cycling. Aggressive power management can produce opposite effects, e.g., hot spot prevention at the cost of increased thermal cycling. Higher integration forces to extend through the third dimension (3D circuits) which in turn increases the thermal problem since the ratio of surface-area/energy significantly worsens. Devices will be exposed to higher temperatures and increase, among others, aging effects. In addition, transient faults increase.

Soft Errors

The susceptibility of switching devices in the nano age against soft errors will increase about 8% per logic state bit for each technology generation, as recently forecasted. Soft errors are caused by energetic radiation particles (neutrons) hitting silicon chips and creating a charge on the nodes that flips a memory cell or logic latches.

The idea of this book has its origin in several international programs on dependability/reliability:

– The SPP 1500 Dependable Embedded Systems program (by DFG of Germany);
– The NSF Expedition on Variability (by NSF of USA); and
– The Japanese JST program.

While this book is not a complete representation of all of these programs, it does represent all aspects of the SPP 1500 and some aspects of the NSF Expedition on Variability and the Japanese JST program.

The book focuses on cross-layer approaches, i.e., approaches to mitigate dependability issues by means and methods that work across design abstraction layers. It is structured in the main six areas "Cross-Layer from Operating System to Application," "Cross-Layer Dependability: From Architecture to Software and Operating System," "Cross-Layer Resilience: Bridging the Gap between Circuit and Architectural Layer," "Cross-Layer from Physics to Gate- and Circuit-Levels," and "Cross-Layer from Architecture to Application." Besides, it contains a chapter in the so-called RAP model: the resilience articulation point (RAP) model aims to provision a probabilistic fault abstraction and error propagation concept for various forms of variability-related faults in deep submicron CMOS technologies at the semiconductor material or device levels. RAP assumes that each of such physical faults will eventually manifest as a single- or multi-bit binary signal inversion or out-of-specification delay in a signal transition between bit values.

The book concludes with a perspective.

We want to thank all the authors who contributed to this book as well as all the funding agencies that made this book possible (DFG, NSP, and JST).

We hope you enjoy reading this book and we would be glad to receive feedback.

Karlsruhe, Baden-Württemberg, Germany Jörg Henkel

Irvine, CA, USA Nikil Dutt

Contents

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

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-

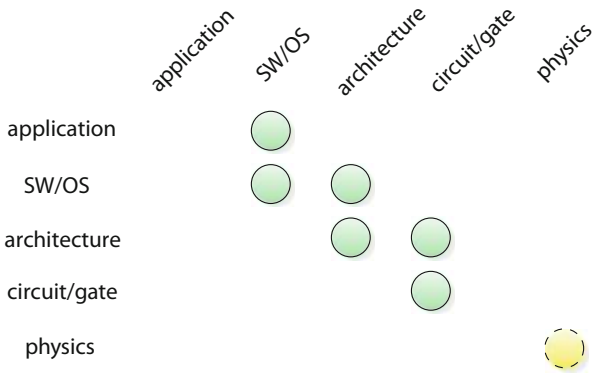


Fig. 1 RAP covers probabilistic error modeling and propagation of physics induced variabilities from circuit/logic up to application level

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

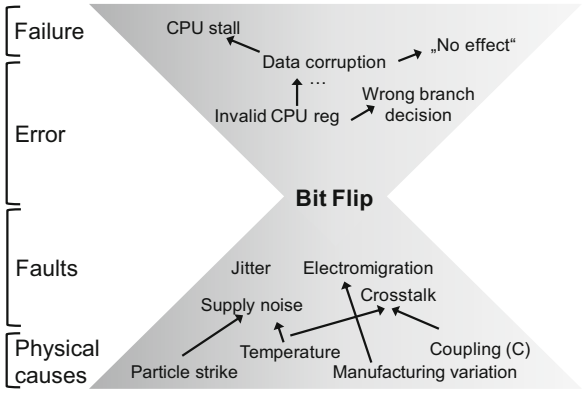


Fig. 2 Fault, error, and failure representations per abstraction levels

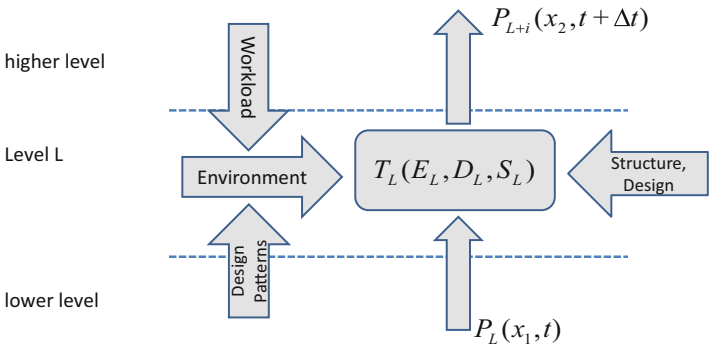


Fig. 3 Error transformation function depending on environmental, design, and system state conditions

$\mathcal{T}_L$ 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

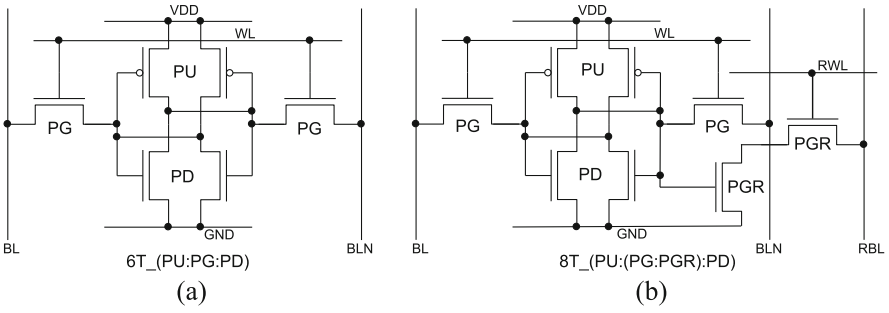


Fig. 4 Circuit schematics for standard 6T (**a**) and 8T (**b**) SRAM bit cells

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].

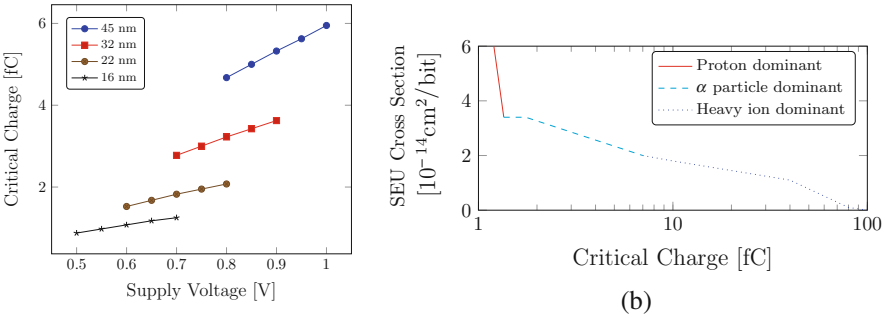


Fig. 5 Technology influence on SRAM bit flips: (**a**) Critical charge dependency on technology node and supply voltage for 6T SRAM cell, (**b**) Particle dominance based on critical charge (adapted from [10])

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.

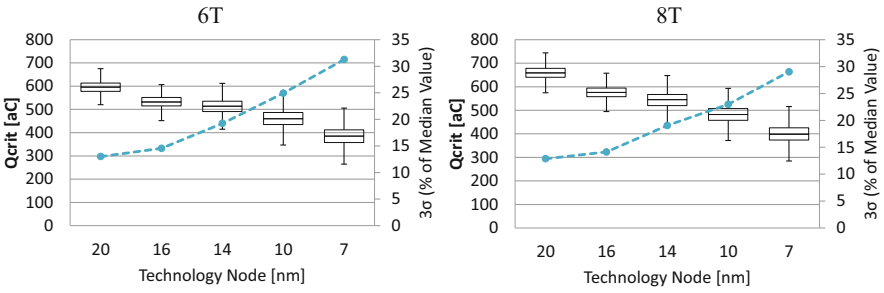


Fig. 6 Qcrit results for a 6T_(1:1:1) high density (left) and an 8T_(1:(1:1):1) (right) SRAM cell

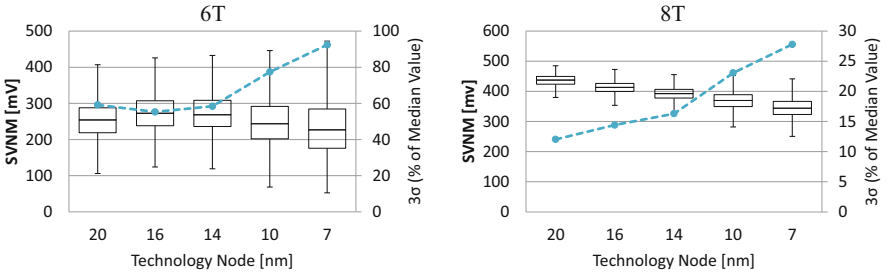


Fig. 7 SVNM results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

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.

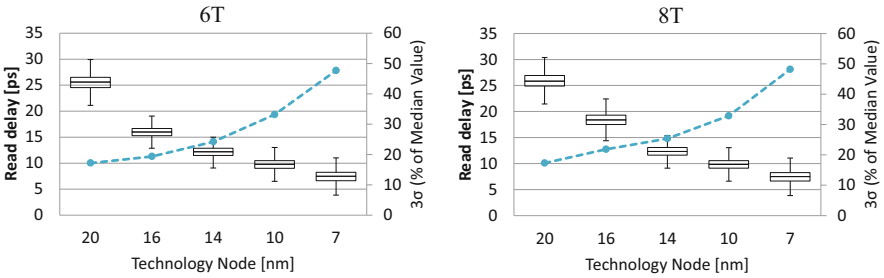


Fig. 8 Read delay results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

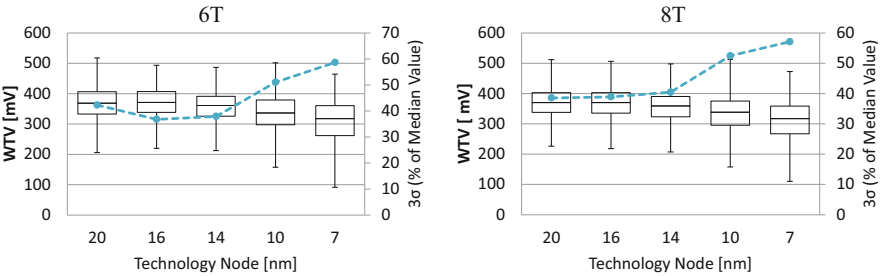


Fig. 9 WTV results for a 6T_(1:1:1) and an 8T_(1:(1:1):1) SRAM cell

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

$$P_{\text{write\_error}}(V_s < V_{swing\_min}) = \int_0^{V_{swing\_min}} f_{V_s}(V)dV \tag{8}$$

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

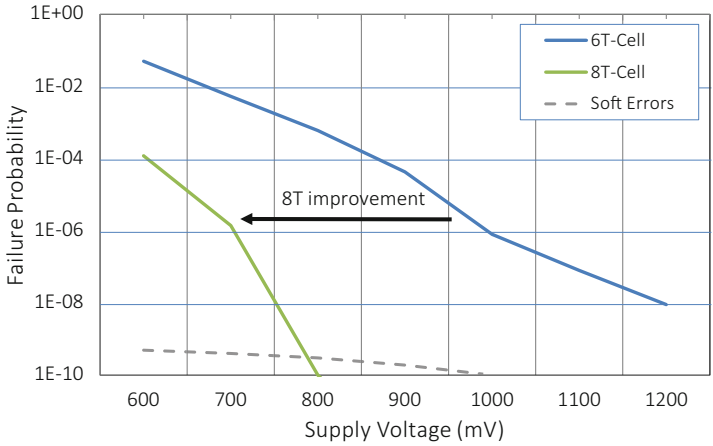


Fig. 10 Memory failure probability (65 nm technology) [1]

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.

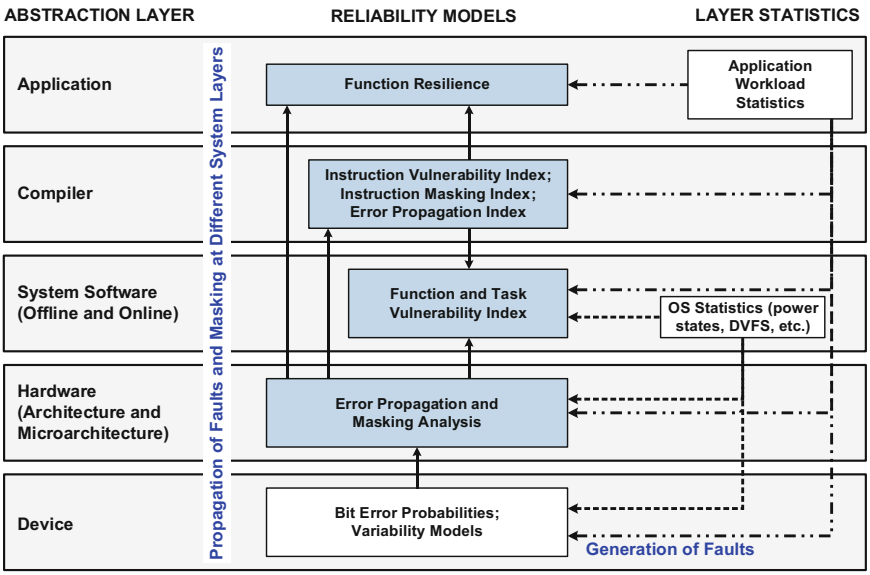


Fig. 11 Cross-layer reliability modeling and estimation: an instantiation of the RAP model from the application software's perspective

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

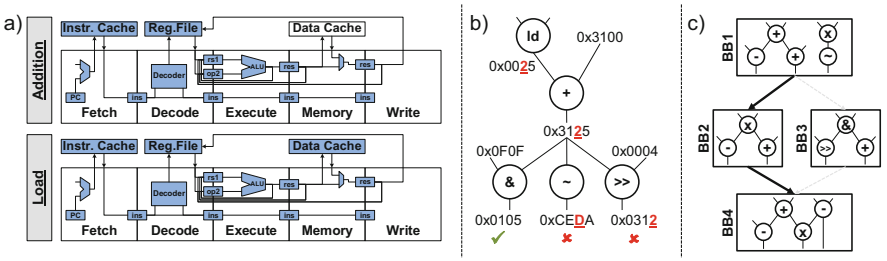


Fig. 12 (**a**) Temporal and spatial vulnerabilities of different instructions executing in a processor pipeline; (**b**) Examples for error propagation and error masking due to data flow; (**c**) Example for error masking due to control flow

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

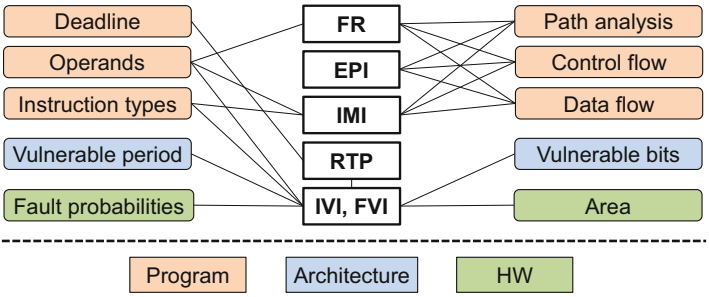


Fig. 13 Composition and focus of the different modeling approaches

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

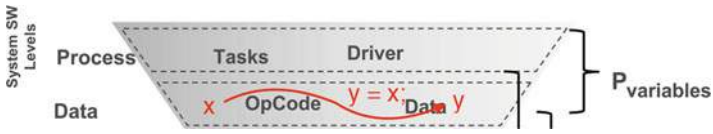


Fig. 14 Horizontal propagation of an error in the RAP model

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.

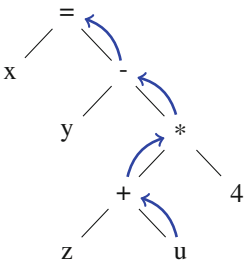
Using a bisecting approach, this results in a binary classification of the worst-case error impact of a data object on a program's execution. If an error in a data object could result in an application crash, the related piece of data is to be marked as critical to the system stability. An example for 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

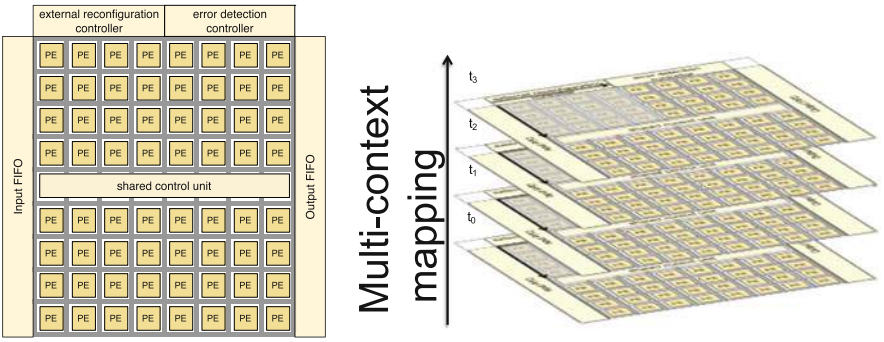


Fig. 16 General DRP structure (left) and temporal application mapping in DRPs (right)

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.

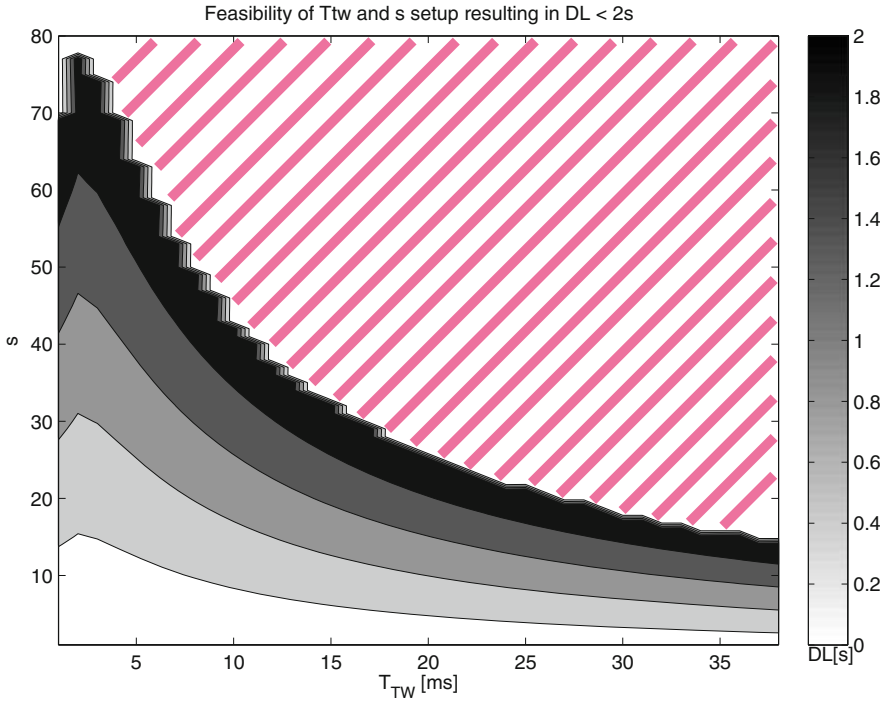


Fig. 17 Feasibility region for an error to be detected within 2 s, with $N = 4$ running at 100 MHz, an observable error probability of $P = 10^{-5}$, a reconfiguration and setup overhead of 1 ms and different scaling factors s and time windows T_{TW}

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).

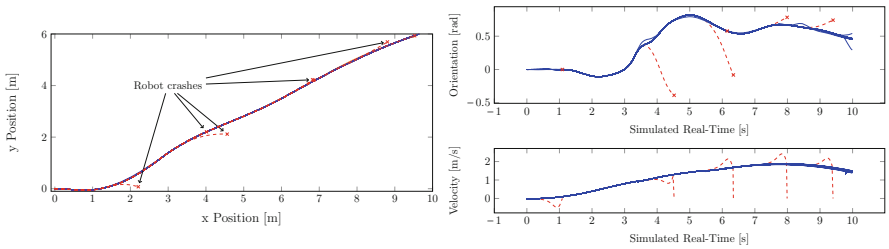


Fig. 18 Robot movement in $x - -y$ plane together with velocity and orientation angle. Dashed lines indicate crashes by CPU stalls

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)

Part I
Cross-Layer from Operating System to Application

Horst Schirmeier

In the embedded systems domain—particularly in highly competitive areas such as the automotive industry—per-device monetary cost reduction is often a first-class optimization goal. The prevailing approach to keeping costs low is to implement more and more functionality in software instead of hardware, with side effects also on other non-functional properties such as total system weight, energy consumption or in-system upgradability. However, this trend towards software shines in the disconcerting light of ever-increasing soft-error rates: As the industry moves towards single-digit nanometer semiconductor structure sizes, the circuits' susceptibility to soft errors continuously increases—unless countered with costly hardware measures that can diminish the gains achieved through scaling.

In consequence, embedded software must cope with increasingly unreliable hardware. In the stack of software layers, the operating system is the layer closest to the hardware, and figuratively the first line of defense against soft errors that are on their way of propagating to the application layer. But as software countermeasures against unreliable hardware tend to be even more costly than their hardware counterparts, they must be used sparingly and tailored for a particular application and use-case scenario.

The three chapters within this area of the book address this cross-layer connection between the operating system and the application layer, and offer different approaches of constructing a reliable system from unreliable hardware components that exhibit erroneous behavior triggered by soft errors.

The first chapter by Engel and Marwedel addresses the problem of error-correction overhead in the context of embedded real-time systems (chapter "Soft Error Handling for Embedded Systems using Compiler-OS Interaction"). The described FEHLER approach introduces *error semantics*, which provides information about the criticality of data objects. A combination of explicit source-code

H. Schirmeier (✉)
Embedded System Software Group, TU Dortmund, Dortmund, Germany
e-mail: horst.schirmeier@tu-dortmund.de

annotations and static data-flow analysis, which derives information for other, non-annotated variables, is used on the application layer to generate a database that allows to classify data objects at runtime. The gathered information is handed across layers to the operating system: When an error is detected—by a mechanism outside of FEHLER's scope—the proposed system-software stack can use this classification database to assess the actual damage, and to flexibly choose one of several possible error-handling methods. Among the information incorporated in this decision is the current real-time scheduler's state, e.g., whether immediately scheduling an error-correction task would cause a deadline miss and should therefore be either delayed or completely skipped. This chapter also introduces the concept of the *Reliable Computing Base* (RCB), a part of the system that is critical in ensuring that the error-handling mechanism is effective, and that must not be affected by soft errors itself.

The second chapter, contributed by Rambo and Ernst, describes how to achieve the goal of application-specific and selective fault tolerance at a much coarser granularity (chapter "ASTEROID and the Replica-Aware Co-scheduling for Mixed-Criticality"). Set in a scenario with a given set of mixed-critical applications, the ASTEROID approach requires a manual criticality classification of application tasks—information that, similar to the FEHLER approach, is passed cross-layer from the application to the operating system. Critical tasks are executed redundantly by the *Romain* system service, exploiting future manycore platforms for the increased system load, and coexist with non-critical tasks. Unlike FEHLER, the ASTEROID approach also comprises a concrete error-detection solution: a microarchitecture-level pipeline fingerprinting mechanism that allows *Romain* to compare replicas with low overhead, facilitated through a cross-layer design involving both the hardware and operating-system layers. Quantifying the minimized overhead, the chapter puts a special focus on the performance of replicated execution, introducing a replica-aware co-scheduling strategy for mixed-critical applications that outperforms the state of the art.

The third chapter by Schirmeier et al. describes the DanceOS approach, focusing on application-specific operating-system construction techniques, similar to FEHLER aiming at fine-grained fault-tolerance approaches (chapter "Dependability Aspects in Configurable Embedded Operating Systems"). The chapter first investigates the general reliability limits of static system-software stacks, and demonstrates a technique to reduce the proverbial "attack surface" of a newly constructed, AUTOSAR-compliant operating system by exploiting knowledge from static task descriptions. By additionally applying classic fault-tolerance techniques to the remaining dynamic kernel data structures, the DanceOS approach yields a highly reliable software system. The second part of the chapter addresses the problem how a pre-existing, legacy dynamic operating-system codebase can be hardened against soft errors in an application-specific way. Using programming-language and compiler-based program transformation techniques—in particular aspect-oriented programming—this part shows how generic fault-tolerance mechanisms can be encapsulated in separate modules, and applied to the most critical data structures identified, e.g., by fault-injection experiments. In both operating-system scenarios—

and similar to FEHLER and ASTEROID—the application drives the fault-tolerance hardening process: In the AUTOSAR-compliant static OS scenario, statically known structural application knowledge is handed to an operating-system tailoring and minimization process; in the dynamic legacy-OS scenario, the application's runtime behavior while being exposed to injected faults provides the information relevant for targeted, selective fault-tolerance hardening. In the third and last part, the chapter expands the considered fault model to whole-system power outages, and demonstrates that persistent memory—combined with transactional memory—can be used for state conservation.

To conclude, all three chapters share the common insight that a cross-layer combination of application layer knowledge and operating-system layer fault tolerance—in the case of ASTEROID additionally involving the hardware layer—enables overhead minimization and optimal, application-specific hardening against soft errors.

Soft Error Handling for Embedded Systems using Compiler-OS Interaction

Michael Engel and Peter Marwedel

1 New Requirements for Fault Tolerance

The ongoing downscaling of semiconductor feature sizes in order to integrate more components on chip and to reduce the power and energy consumption of semiconductors also comes with a downside. Smaller feature sizes also lead to an increasing susceptibility to *soft errors*, which affect data stored and processed using semiconductor technology. The amount of disturbance required to cause soft errors, e.g. due to the effects of cosmic particles or electromagnetic radiation on the semiconductor circuit, has declined significantly over the last decades, thus increasing the probability of soft errors affecting a system's reliable operation.

2 Semantics of Errors

Traditionally, system hardening against the effects of soft errors was implemented using hardware solutions, such as error-correcting code circuits, redundant storage of information in separate memories, and redundant execution of code on additional functional units or processor cores. These protection approaches share the property that they protect all sorts of data or code execution, regardless of the requirement to

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

P. Marwedel
Department of Computer Science, TU Dortmund, Dortmund, Germany
e-mail: peter.marwedel@tu-dortmund.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_2

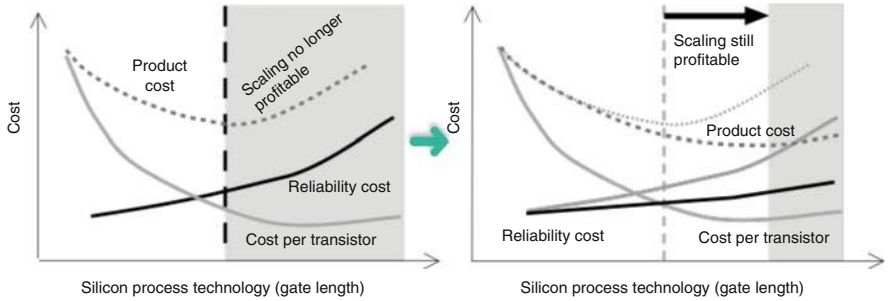


Fig. 1 Enabling profitable scaling using software-based fault tolerance [1]

actually enforce protection. In other terms, they do not possess knowledge about the *semantics* of data and code related to the reliable operation of a system.

As shown by Austin [1], reproduced on the left-hand side of Fig. 1, together with a rising probability of soft errors, this results in a significantly increasing overhead in hardware required to implement error protection. As technology progresses, at a certain point in time, the cost of this overhead will exceed the savings due to the utilization of more recent semiconductor technologies, resulting in diminishing returns that render the use of these advancements unattractive.

The fundamental idea applied by the FEHLER project is to reduce the amount of error handling required in a system by introducing semantic knowledge. We enable a system's software to dynamically decide at runtime whether an error that occurred is critical to the system's operation, e.g. it might result in a crash in the worst case, or is not critical, e.g. an error might only result in an insignificant disturbance of a system's output. In turn, this enables the system to handle only *critical errors* and ignore the others. This *flexible error handling* results in a significantly reduced hardware overhead for implementing fault tolerance, which leads to an increased profitability window for semiconductor scaling, as shown on the right-hand side of Fig. 1.

One important consideration when designing such a selective approach to fault tolerance is which components of a system actually have to be protected from errors. Inspired by the concept of the trusted computing base in information security, we introduced the *Reliable Computing Base* (RCB) [6] to indicate the hardware and software components of a system that are critical in ensuring that our flexible error handling approach is effective.

Accordingly, we define the RCB as follows:

> The Reliable Computing Base (RCB) is a subset of software and hardware components that ensures the reliable operation of software-based fault-tolerance methods. Hardware and software components that are not part of

(continued)

the RCB can be subject to uncorrected errors without affecting the program's expected results.

To design efficient fault-tolerant systems, it is essential to *minimize* the size of the reliable computing base. In the case of FEHLER, this implies that the number and size of hardware and software components required to ensure that upcoming *critical errors* will be corrected are to be reduced as far as possible.

Commonly, code-based annotations such as [13] are used to indicate sections of code to be protected against errors regardless of the data objects handled by that code. This implies an overhead in runtime—protection of the executed section of code applies to all its executions without considering its execution semantics—as well as in programmer effort, since error propagation analyses using control and data flow information would have to consider all data objects handled in annotated code sections. In order to increase the efficiency of this approach, additional manual annotations seem indispensable.

A more efficient approach from a software point of view is to identify the minimal amount of *data objects* that have to be protected against soft errors. Data flow analyses provided by FEHLER allow to determine the worst-case propagation of errors throughout a program's execution, thus determining the precise set of data objects requiring protection against errors. Additional savings at runtime are achieved by employing a microkernel system tailored to exclusively address error handling, leaving the remaining operating system functions to a traditional embedded kernel running on top of it. An analysis of the possible savings for a real-world embedded application is given in Sect. 7.

3 FEHLER System Overview and Semantic Annotations

Based on the observations described above, one central objective of the FEHLER system is to enable the provision of semantics describing the worst-case effects of errors on data objects.

Commonly, the hardware of a system only has very limited knowledge about the semantics of data that it processes.[1] More semantic information, such as the types of data objects, is available on the source code level. However, this information is commonly discarded by the compiler in later code generation and optimization stages when it is no longer required to ensure program correctness. Some of this information can already be utilized to provide error semantics. For example, pointer

[1] For example, a processor could distinguish between integer and floating point data due to the use of different registers and instructions to process these, but a distinction between pointer and numeric data is often not possible on machine code level.

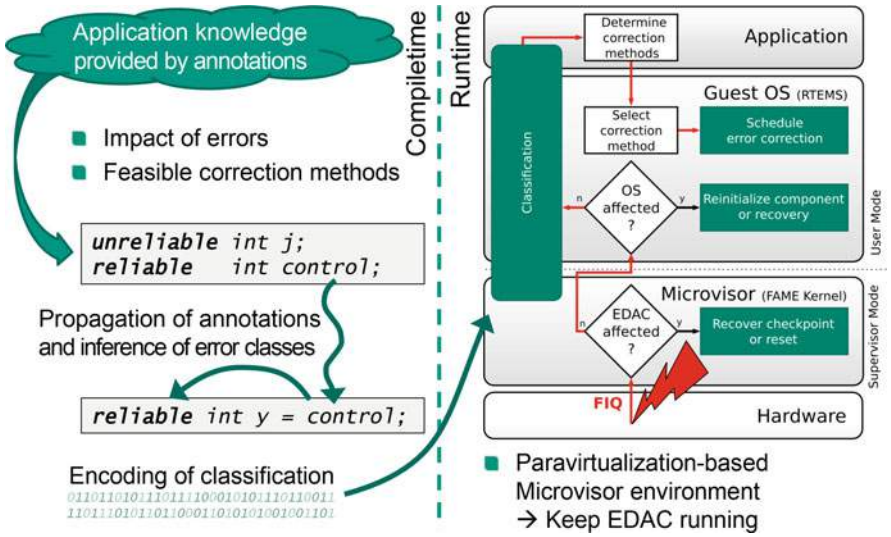


Fig. 2 Interaction of compile time and runtime components of FEHLER

data types and variables influencing the control flow (e.g. used to conditionally execution of code or control loops) are deemed essential in ensuring correct program execution. Accordingly, *static analyses* performed during compile time are able to extract this information.

However, additional information about the relevance of data with regard to the correct behavior of a system in its intended context, e.g. in embedded systems where an incorrect output controlling a peripheral might result in damaging effects, is not expressed *explicitly* in the code. Hence, we have to provide additional information in order to enable static analyses to derive more information about the worst-case criticality of a data object.

This additional semantic information allows the system to classify errors. Data objects which are deemed critical to a program's execution, i.e. may cause the program to crash, are annotated with a `reliable` type qualifier. All objects for which errors in data will result only in an insignificant deviation of the program's behavior in the worst case are provided with an `unreliable` type qualifier.

Classifying data objects into only these two classes is a rather coarse approach. However, as shown later, this minimalistic approach is effective and efficient for systems experiencing normal error rates, i.e. applications not exposed to radiation-rich environments, such as space and aviation systems. Approaches for improved QoS assessment are discussed in Sect. 10.

The interaction of compile time and runtime components of a FEHLER-based system is shown in Fig. 2. Here, the compile time component, realized as a compiler performing static analyses and transformations in addition to code generation, extracts semantic information on the criticality of data objects, analyzes the

program's control and data flow to determine possible error propagation paths and to generate appropriate type qualifiers, and encodes this information along with the generated program binary.

This information lies dormant as long as no error affects the system's operation. Since error detection is outside of the scope of FEHLER, the system is prepared to interface with a number of different error detection methods. In the example in Fig. 2, we assume that a simple hardware mechanism, such as memory parity checks, is employed. When an incorrect parity is detected during a memory access, a special interrupt is raised that informs the system of the error.

Here, our runtime component, the Fault-aware Microvisor Environment (FAME) [11] comes into play. FAME is intentionally restricted to only provide functionality that enables decisions about the necessity of error handling, relegating all other functionality typically found in system software to components outside of the microkernel. This reduced functionality is an additional contribution to RCB minimization. FAME provides a handler for the given error signalization method, which is able to determine the *address of the affected memory location*. As soon as the microkernel is able to ensure that itself is not affected, which can be ensured by RCB analysis and minimization, it determines whether the embedded OS kernel running on top or the application is affected. If this is the case, error handling is initiated. In case of an error affecting the application, FAME consults the semantic information provided by the compile time components and determines if error correction is required or if the error can be safely ignored. Further details of FAME are described in Sect. 6.

Like error detection, specific correction methods are not the focus of FEHLER. Instead, FEHLER is enabled to interface with different standard as well as application-specific correction methods. An example for a standard error correction would be the application of checkpointing and rollback. An application-specific method would be a function that corrects an affected macro block in a video decoder by interpolating its contents from neighboring blocks instead of redecoding the complete video frame, thus saving a considerable amount of compute time.

4 Timing Behavior

Figure 3 shows possible scheduling orders in case of a detected error. In an approach that neglects to use criticality information ("naive approach"), the detection of an error implies an immediate correction action in hardware or software. This potentially time-consuming recovery delays the execution of subsequent program instructions, which may result in a deadline miss.

The flexible approach enabled by FEHLER allows the system to react to an error in a number of different ways. Here, the classifications described above come into play. Whenever an error is detected, the system consults the classifications provided alongside the application ("C" in Fig. 3). This lookup can be performed quickly

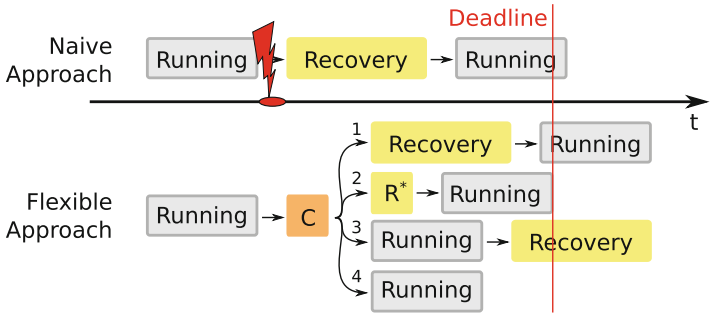


Fig. 3 Different scheduling possibilities for error handling

and provides information on how to handle the error at hand. Specifically, it can be determined *if*, *how*, and *when* the error needs to be corrected:

- *if:* Whether errors have to be handled or not depends mainly on the error impact. If an error has a high impact, error recovery will be mandatory. In contrast, if an error only has a low impact at all, e.g. the color of a single pixel in a frame buffer is disturbed, further handling can be omitted. Handling errors in the latter case will improve the quality of service at the cost of requiring additional compute time. Error impacts are deduced using static analysis methods as described below.
- *how:* Error handling depends on the available error correction methods, the error impact, and the available resources. In FEHLER, commonly a bit-precise correction method such as checkpoint-and-recovery as well as an "ignore" method (case 4 in Fig. 3) doing nothing is available. In addition, the programmer can provide application-specific correction methods, denoted by "R*". Such a method may be preferable, since it can be faster than the generic correction method provided.
- *when:* Error scheduling can decide when an error correction method has to be scheduled. In a multitasking system, often, the task with the highest priority is executed. Hence, if a high priority task is affected, error correction has to be scheduled immediately (cases 1 and 2). If a low priority task is affected, the high priority task can continue execution and the error handling will be delayed (case 3). In order to enable the mapping of errors to different tasks, a subscriber-based model can be employed [12].

Overall, this flexibility allows a system to improve its real-time behavior in case of errors. While this may not be acceptable for hard real-time systems, the behavior of soft real-time applications, such as media decoding, can be significantly improved. The example of an H.264 video decoder is used in Sect. 7 to evaluate the effectiveness and efficiency of FEHLER.

To enable the flexible handling of errors at runtime, the runtime system requires the provision of detailed, correct meta information about the data objects in the given

application. The static analyses employed to obtain this information are described in the following section.

5 Static Analyses

The correct determination of all data objects critical to a program's execution, which form part of the RCB, is crucial to ensure that errors threatening to result in a system crash are corrected before they affect its operation.

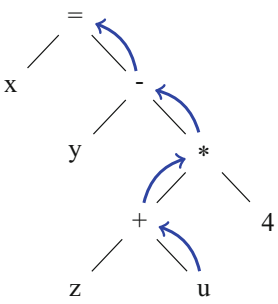
Our static analysis is based on the concept of subtyping [7]. In FEHLER, additional semantic information on the criticality of a data object to the application's stability is provided by extending the C language using *reliability type qualifiers*. These qualifiers enable a developer to indicate to the static analysis stages whether a data object is deemed critical to a program's execution (`reliable` classification) or if errors in data will result only in an insignificant deviation of the program's behavior in the worst case (using the `unreliable` type qualifier).

Accordingly, we have to ensure that reliable data objects must not be modified unpredictably to guarantee that the application will not crash. In contrast, the application can tolerate deviations from the expected values in unreliable data objects.

Rules for the use of our new type qualifiers applied by our static code analysis fall into two groups: *prohibit* and *propagation rules*. Prohibit rules ensure that operations on the annotated data objects are executed error-free, whereas propagation rules reflect the possible propagation of errors from an affected data object to others throughout the control and data flow.

Errors in certain data objects may result in a large deviation in the control flow or even an unintended termination of the application. Prohibit rules ensure that those data objects are annotated with the reliable type qualifier; accordingly, errors affecting that data are classified as fatal errors. Data objects serving as a reference to a memory address, i.e. pointers in C, are an important example for this. An error in a pointer that is used for reading will result in either a different address that is read, possibly resulting in the loading of a completely unrelated data object, or even an access to a non-existing memory location, resulting in a processor exception that terminates the application. Pointers used for writing data can result in correct data being written to an unintended memory location, resulting in unexpected error propagation that is especially hard to diagnose. Indexes for arrays behave in a similar way, resulting either in a write to a different array element or, due to the lack of bounds checking for array indexes in C, a write to an arbitrary memory location. Other critical data types include controlling expressions for loops and conditional statements, divisors, branch targets, and function symbols. For details, we refer the reader to the description in [15].

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




```
...

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

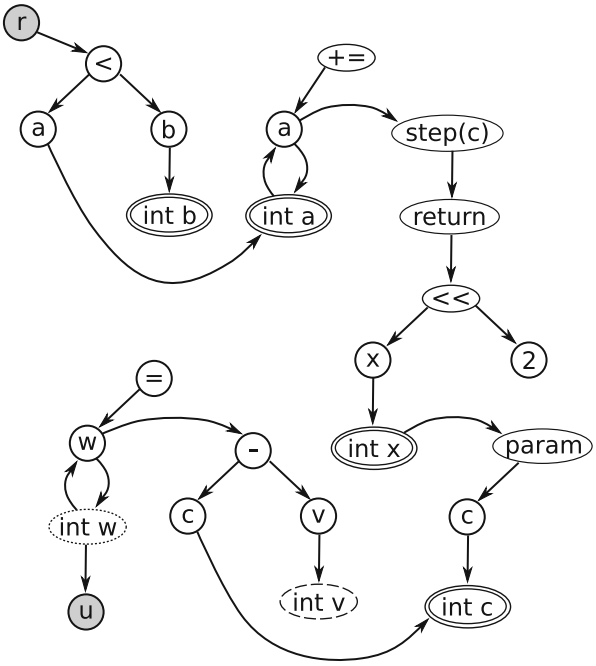
Listing 1 Data flow analysis of possible horizontal error propagation and related AST representation

The content of a data object annotated as unreliable may be affected by an uncorrected error. In turn, that error can propagate to other data objects whenever its content is copied or used in an arithmetic or logic expression, as shown in Listing 1. Here, the curved arrows indicate that an error can propagate from one subexpression to the following along the edges of the syntax tree. Accordingly, the content of a resulting data object cannot be considered reliable and thus has to be qualified as unreliable. The dependencies between type qualifiers of different data objects are modeled by the FEHLER propagation rules. In addition to calculations and assignments, other uses of data objects affected by error propagation are the copying of parameters to functions using call-by-value semantics and cast expressions.

```
int step(int x) {
    return x << 2;
}

void main(void) {
    int a, b, c;
    unreliable int w;
    int v;

    // Initializations
    // omitted for brevity
    while (a < b)
        a += step(c);

    w = c - v;
}
```

Listing 2 Code example and related type deduction graph

Propagation rules not only help in detecting erroneous data flow from unreliable to reliable data objects, but also reduce the overhead required by the programmer



to create annotations. Since it is unrealistic to expect that each and every data object in a complex program will be annotated manually, our static analyses help to deduce correct type qualifiers for unannotated data. This deduction is enabled by the construction of a *type deduction graph* (TDG), as shown in Listing 2. Here, the shaded special vertices marked ⓡ and ⓤ represent an enforcement of the type qualifiers reliable and unreliable by prohibit rules or explicitly stated annotations. The set of edges of the TDG then reflects the dependencies between the type qualifiers, data objects, operations, and assignments.

```
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 3 Invalid assignments

Accordingly, the use of the TDG enables the compiler to flag invalid data propagation from unreliable to reliable data objects. An example containing a number of such invalid propagations is given in Listing 3.

Overall, the static analyses provided by the FEHLER compiler toolchain enable programmers to state reliability requirements that cannot be deduced from the program itself while ensuring that these manual annotations do not accidentally provide a way to propagate unreliable data to reliable data objects. During runtime, the annotations are then used to enable flexible error handling by allowing the operating system to ignore errors in data objects marked as unreliable, thus enabling a tradeoff between the obtained quality of service and the required error correction overhead, e.g. in terms of time or energy.

6 FEHLER Runtime System

Viewed from the top, as shown in Fig. 6, an application with integrated classification information is running on a virtualized guest OS. The guest OS is linked against the FAME Runtime Environment (FAMERE). FAMERE is responsible for the flexible error handling as well as the interfacing with the microvisor. The microvisor runs low-level error correction and ensures the feasibility of software-based error handling (Fig. 4).

The FAMERE runtime is based on our specialized microvisor component which has control over the hardware components relevant to error handling. The main purpose of the microvisor is to isolate critical system components from possible error propagation and schedule the error handling if required. Critical components in this context are resources required to keep error detection and correction running. Depending on the underlying hardware, the actual critical resources vary. If, for example, errors are signaled via interrupts, the interrupt controller will be an element of the critical resource set.

Since the microvisor itself can be affected by errors, it is considered to be a part of the RCB. The microvisor is incapable of protecting itself, since it implements the basic error handling routines. In order to ensure the effectiveness of error

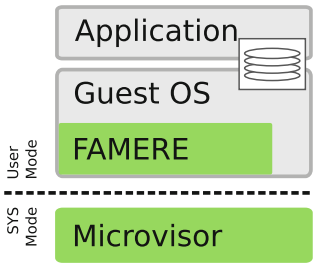


Fig. 4 The runtime software stack of FEHLER. The microvisor is only involved in case of an error, whereas all other resources are administered by the paravirtualized guest OS. The guest OS is extended by FAMERE, the system component responsible for evaluating compiler-provided information on the criticality of errors

handling, fault-free hardware components are required to execute the software-based fault-tolerance mechanisms. In turn, these hardware components also have to be considered part of the RCB. Reducing the code and data size of the microvisor itself is, thus, an optimization objective required to reduce the overall size of the RCB.

To shield the RCB from error propagation, our microvisor uses paravirtualization [16]. The microvisor is tailored to the needs of embedded systems and fault tolerance. To keep the virtualization overhead low, it supports only a single guest operating system. This removes the requirement to provide virtual CPUs and CPU multiplexing. In addition, caches and TLB entries need not be switched between different guest OS instances. An additional responsibility of the microvisor is the creation of full system checkpoints. These are used to restore a valid system state in case of a severe error affecting the FAMERE runtime. FAMERE is a library in the guest OS that combines compile and runtime information required to implement flexible error handling [12].

Error handling is the central task of FAMERE. Figure 5 gives a detailed view of the error-handling procedure at runtime (the right-hand side of Fig. 2). In order to enable a prioritization of error handling, tasks affected by an error in the OS running on top of the microvisor have to be identified. FAMERE determines affected tasks using a memory subscriber model [12] in which tasks explicitly subscribe to and unsubscribe from data objects prior resp. after their use. Accordingly, each data object is annotated with a set of tasks currently using the object, enabling FAMERE to assign a memory address to the set of tasks using the address at the current moment.

If there are higher prioritized tasks not affected by current error, further error handling will be delayed until all higher prioritized tasks finish execution. Error classification will then be performed when the error handling is scheduled again by the microvisor, thus minimizing the impact on system timing when an error occurs.

Together with classification information for data objects, our microvisor and the FAMERE library enable the FEHLER system to implement the envisioned flexible error-handling principles. By keeping the amount of functionality and the related code and data sizes of the microvisor low, the RCB size could be reduced significantly.

7 Use Case: A Fault-Tolerant QoS-Aware Soft Real-time Application

In order to assess the effectiveness and efficiency of the selective error correction approach enabled by FEHLER, we analyzed typical embedded applications in the presence of errors. Since microbenchmarks only tend to give a restricted view of the effects of errors, we used a real-world application to evaluate the possible reduction in overhead.

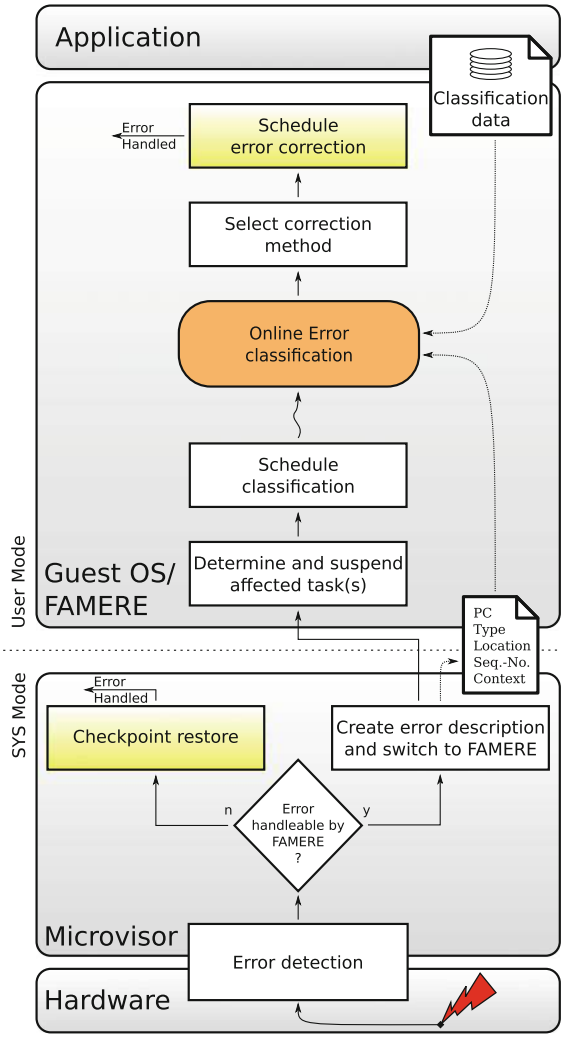


Fig. 5 Error handling in the runtime software stack of FEHLER. If an error is signaled (red flash symbol), the microvisor checks whether the fault affects the RCB. If the RCB is affected, the microvisor automatically restores the last system checkpoint. Otherwise, error handling is delegated by sending a message to FAMERE, which includes an error description containing information about the occurred error as well as the user space context

As mentioned above, the class of applications that we expect to benefit most from our flexible error-handling approach are soft real-time applications that are able to accept—or even make use of—varying levels of QoS in their output. Thus, we used a constrained baseline profile H.264 video decoder application comprising ca. 3500 lines of ANSI C code as a real-world benchmark to assess the effectiveness and efficiency of FEHLER [9].

The evaluation is performed on a simulated embedded system using Synopsys' CoMET cycle-accurate simulator as well as a physical platform based on a Marvell ARM926-based SoC. CoMET is configured to resemble the real system by simulating a 1.2 GHz ARM926 system with 64 MiB RAM, 16 MiB ROM, and 128 KiB

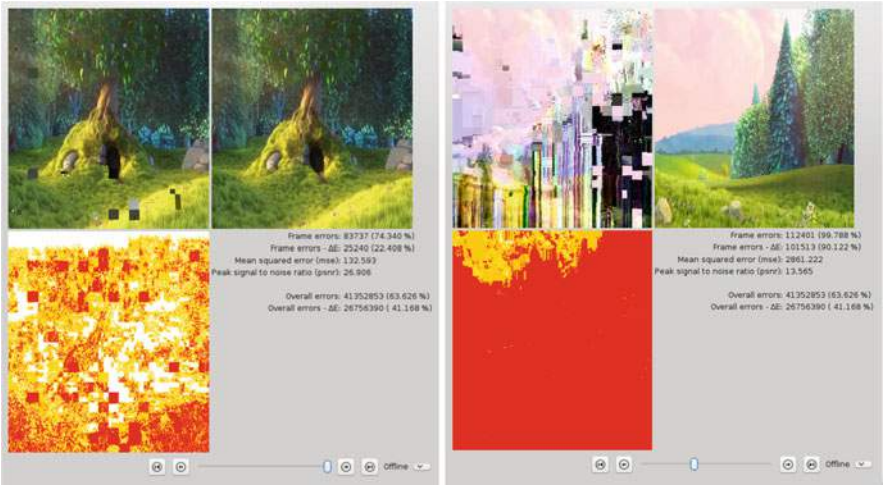


Fig. 6 Analysis of error impacts on the H.264 video decoder using different injection rates

reliable RAM (using ECC-based hardware error protection). All components are considered reliable, except the 64 MiB of RAM.

The H.264 video decoder is configured to create a checkpoint after every displayed frame. In each experiment, we decoded 600 frames in total at a rate of 10 frames per second and a resolution of 480×320 pixels.[2]

We were primarily interested in evaluation results showing the impact of the injected errors themselves on the achievable QoS of the decoded video, the possible reduction of the RCB size using flexible error handling as well as the impact of error handling on system timing.

To assess the impact on the QoS, we developed the quality assessment tool shown in Fig. 6 [8]. It receives video frames decoded by the target ARM system under the influence of errors using FEHLER's flexible error correction and compares these frames to the correctly decoded reference frames (indicated by the yellow and red squares in the lower left pictures–the more red, the larger the difference between the two frames is). For each frame, the tool then calculates several different metrics indicating the QoS, e.g. the peak signal-to-noise ratio (PSNR) and the ΔE color distance metrics. The left-hand side of Fig. 6 shows a moderate error injection rate resulting in some visible defects in the output, whereas the right-hand side shows an artificially high injection rate which renders the output unusable.

For evaluation, we injected uniformly distributed transient faults into RAM. For each memory access, error detection in hardware is simulated. If the processor

[2] Although resolution and frame rate seem rather low, this setup leads to a CPU utilization of more than 65%, since we decode H.264 in software only. However, higher resolutions and frame rates will be possible if more computing power is available.

Table 1 Peak Signal-to-Noise Ratio (PSNR) for different error injection rates [dB]

	$\lambda = 1 \cdot 10^{-16}$	$\lambda = 1 \cdot 10^{-15}$	$\lambda = 1 \cdot 10^{-14}$
All errors handled	36.19	36.15	n/a
Flexible error handling	36.19	36.18	29.01
Flexible + application-specific	36.20	36.12	28.95

accesses an erroneous word, an interrupt will be raised. The number of faults to be injected is determined by a Poisson distribution with a configurable parameter λ.[3]

Table 1 shows QoS results given as PSNR values for different injection rates and correction approaches. We compare a standard correction approach–correcting all errors irrespective of the worst-case outcome–with two approaches based on FEHLER, one which only uses generic error correction such as checkpoint-and-restore and one which, in addition, applies more efficient, application-specific error correction methods. It can be seen that for low error injection rates ($\lambda = 1 \cdot 10^{-16}$ and $1 \cdot 10^{-15}$), uncorrected errors result in a PSNR of about 36 dB, which is still a reasonable quality for lossy compressed media and is similar to the quality of VHS video. For the high error rate ($\lambda = 1 \cdot 10^{-14}$), however, the PSNR drops below 30 dB.[4]

It is important to notice that, although high injected error rates can lead to a significant degradation of the perceived QoS, the primary objective of the binary classification of error impacts employed by FEHLER is achieved—we were unable to provoke the system to crash no matter what the used error injection rate was.

Based on the configuration described above, we analyzed the fraction of memory that the compiler annotated as `unreliable`, implying no protection against errors is required. This fraction is a direct indicator of the reduction of the size of memory that has to be protected, i.e., the RAM memory component of the RCB. In traditional software-based error correction approaches, all of the RAM would be considered part of the RCB. Table 2 shows the results of this evaluation for different video resolutions. It can be observed that for low resolutions, the amount of data classified as `reliable` dominates the memory usage. However, the share of this type of memory is reduced when decoding videos with higher resolutions. For a 720p HD video, already 63% of the RAM used by the H.264 decoder can remain unprotected using FEHLER classifications.

The remaining interesting evaluation is the impact of flexible error handling on the soft real-time properties of the video decoder application. In the first two

[3]Not all injected faults are visible by the application, since faults are only detected when the corresponding memory cell is accessed.

[4]To control the amount of faults to inject, a Poisson distribution with configurable parameter λ is used. The time base used for the Poisson distribution is memory bus ticks. Faults are randomly injected and are equally distributed over the memory. Hence, the locations of the accesses have no influence on the fault distribution.

Table 2 Reduction of the amount of reliable memory required by the H.264 video decoder

Video resolution	Memory size of reliable data	Percentage	Memory size of unreliable data	Percentage
176 × 144	90 kB	55%	74 kB	45%
352 × 288	223 kB	43%	297 kB	57%
1280 × 720	1585 kB	37%	2700 kB	63%

Table 3 Average deadline misses for different error-handling configurations

Error rate		Naive error handling		Flexible error handling		Flexible + application -specific	
λ	$[s^{-1}]$	# Avg. Miss	Avg Missed by	# Avg. Miss	Avg Missed by	# Avg. Miss	Avg Missed by
$\lambda = 1 \cdot 10^{-16}$	0.14	0.00	0.00 ms	0.00	0.00 ms	0.00	0.00 ms
$\lambda = 1 \cdot 10^{-15}$	1.44	2.86	8.15 ms	0.52	7.93 ms	0.36	4.89 ms
$\lambda = 1 \cdot 10^{-14}$	35.84	–	–	1937.87	10,268.98 ms	1887.12	9346.16 ms

columns of Table 3, the observed average error rates (of detected faults) are given, ranging from several faults per minute to an artificially high rate of 36 per second.

We analyzed three different scenarios. In naive error handling, the system treats every error as an error which cannot be handled by FAMERE. Hence, a checkpoint is immediately restored. For this scenario, columns three and four in Table 3 show the average amount of missed deadlines and the average duration of a deadline miss, respectively. For the lowest error rate, no deadline misses occur since enough slack time is available for the recovery of checkpoints. If the error rate increases by an order of magnitude, deadline misses can be observed. On average, deadlines were missed by 8.15 ms. For the highest error rate, no run of the experiment terminated within a set limit of 2 h of simulation time, thus no results are given here.

The results for flexible error handling are shown in columns five and six. Here, only errors affecting reliable and live data are handled by checkpoint recovery. Errors affecting other data are ignored. Flexible error handling reduces the number of deadline misses significantly (81.75%). The time by which a deadline is missed is reduced as well (2.70%). For the artificially increased rate of 35.84 errors per second, however, significant deadline misses could be observed.

The final timing evaluation scenario augmented flexible error handling by including an application-specific error-handling method. For data objects with a special annotation, this method is able to transform a corrupted motion vector into a valid state. For these cases, a time-consuming rollback to a valid system state is not required, reducing the overhead for error correction. Accordingly, using this approach, deadline misses could be reduced by 87.37% for the second highest error rate.

To conclude the overview of our evaluation, we provide an overview of a possible use of application-specific error correction approaches for our H.264 video decoder.

	Impact on QoS	Urgency	Fault handling methods
High Impact	Program termination	fix immediately	rollback
	Corrupted frame input	until frame is displayed	redisplay last frame
	Disturbance in frame header	until frame is displayed	rollback, spare frame, redisplay last frame
Medium Impact	Disturbance in DCT coefficients	fix immediately	rollback, ignore
Low Impact	Disturbance in macro block	until frame is displayed	rollback, copy neighbor block, ignore,
	Disturbance in single pixel	until frame is displayed	rollback, copy neighbor pixel, ignore
		⋮	
	No Impact	none	ignore

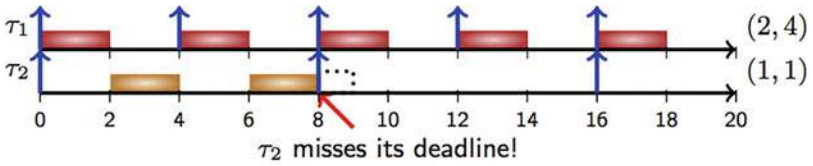
Fig. 7 Classification of error impacts for the H.264 video decoder

As shown in Fig. 7, errors can occur in different data structures, such as frame header or macro blocks. These can be handled by a number of efficient application-specific error correction approaches.

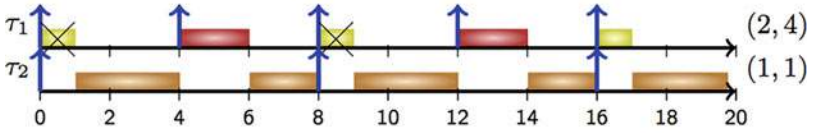
8 Use Case: Adaptive Error Handling in Control Applications

Control-based systems are the basis of a large number of applications for embedded real-time systems. The inherent safety margins and noise tolerance of control tasks allow that a limited number of errors might be tolerable and might only downgrade control performance; however, such limited errors might not lead to an unrecoverable system state. In control theory literature, techniques have been proposed to enable the stability of control applications even if some signal samples are delayed [14] or dropped [2]. Accordingly, we expect that our idea of flexible fault tolerance as described for the video decoder case will also be applicable to control applications.

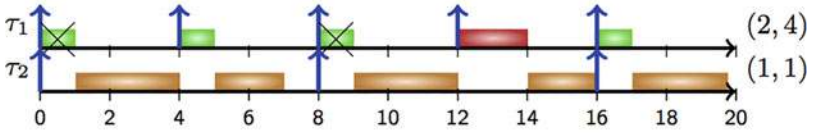
As described above, software-based fault-tolerance approaches such as redundant storage or code execution may lead to system overload due to execution time overhead. For control tasks, an adaptive deployment of related error correction is desired in order to meet both application requirements and system constraints.



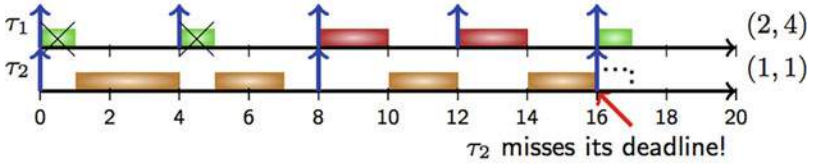
(a) τ_1 is fully protected with reliable versions.



(b) The reliable version of τ_1 is executed with a static distribution trivially.



(c) τ_1 is dynamically compensated with a reliable execution on the forth instance.



(d) τ_1 is not feasible if an additional fault occurring at $t = 4$.

Fig. 8 Different ways to deal with soft errors: red blocks represent reliable executions, green blocks represent executions with error detection, while yellow blocks represent unreliable version without any protection (deadlines are implicit in the schedules shown)

Thus, it has to be investigated how and when to compensate, or even ignore errors, given a choice of different techniques. In an initial case study, we observed that a control task can tolerate limited errors with acceptable performance loss [5].[5]

The general approach used to investigate the effectiveness of this approach is to model the fault tolerance of control applications as a (m, k)-constraint which requires at least m correct runs out of any k consecutive runs to be correct. We investigate how a given (m, k) constraint can be satisfied by adopting patterns of task instances with individual error detection and compensation capabilities. Figure 8 shows four different ways to handle soft errors. Some of the presented schedules are infeasible, since they lead to deadline misses.

[5]This section is based on joint work with Kuan-Hsun Chen, Björn Bönninghoff and Jian-Jia Chen, TU Dortmund.

A static approach to ensure this property is Static Pattern-Based Reliable Execution. In this approach, we enforce the (m_i , k_i) constraints by applying (m, k) static patterns to allocate the reliable executions for task τ_i. While the adopted pattern will affect the schedulability, stability, and flexibility, deciding the most suitable pattern is out of scope of this work.

Due to its inability to react dynamically to changes at runtime, it is obvious that this approach has to be overprovisioning. Thus, we introduce a runtime adaptive approach called Dynamic Compensation that enhances Static Pattern-Based Reliable Execution by recognizing the need to execute reliable instances dynamically instead of having a static schedule.

It is too pessimistic to allocate the reliable instances strictly due to the fact that soft errors randomly happen from time to time. To mitigate the pessimism, we propose an adaptive approach, called Dynamic Compensation, to decide the executing task version on-the-fly by enhancing Static Pattern-Based Reliable Execution and monitoring the erroneous instances with sporadic replenishment counters.

The idea is to execute the unreliable instances and exploit their successful executions to postpone the moment that the system will not be able to enforce an (m, k) constraint, in which the resulting distribution of execution instances still follows the string of static patterns in the worst case.

With Dynamic Compensation, we prepare a mode indicator Π for each task to distinguish the behaviors of dynamic compensation for different status of tasks, i.e., $\Pi \in \{\text{tolerant, safe}\}$. If a task τi cannot tolerate any error in the following instances, the mode indicator will be set to safe and the compensation will be activated for the robustness accordingly. If it can tolerate error in the next instance, the mode indicator will be set to tolerant and execute the unreliable version with fault detection.

Our investigation showed that in embedded systems used for control applications which are liable to both hard real-time constraints and fulfillment of operational objectives, the inherent robustness of control tasks can be exploited when applying error-handling methods to deal with transient soft errors induced by the environment. When expressing the resulting task requirement regarding correctness as a (m, k) constraint, scheduling strategies based on task versions with different types of error protection become applicable. We have introduced both static- and dynamic-pattern-based approaches, each combined with two different recovery schemes. These strategies drastically reduce utilization compared to full error protection while adhering to both robustness and hard real-time constraints. To ensure the latter for arbitrary task sets, a schedulability test is provided formally. From the evaluation results, we can conclude that the average system utilization can be reduced without any significant drawbacks and be used, e.g., to save energy. This benefit can be increased with further sophistication; however, finding feasible schedules also becomes harder.

For an in-depth discussion in the context of a follow-up investigation of this topic, we refer the reader to [17].

9 Application of FEHLER to Approximate Computing

Whereas the work described above concentrates on handling bit flips in memory, more recently, *approximate computing* approaches have been investigated to design energy-efficient systems that trade result precision for energy consumption.

One of the novel semiconductor technologies at the basis of approximate computing is Probabilistic CMOS [3] (PCMOS). Figure 9 shows the general layout of a ripple-carry adder based on PCMOS technology (PRCA). While traditional energy-conserving circuits use uniform voltage scaling (UVOS), PCMOS employs biased voltage scaling (BIVOS), which provides different single-bit full adder components with differing supply voltages that increase from the least to the most significant bit in multiple steps. As a consequence, the delay required to calculate a bit decreases from the LSB to the MSB; accordingly, the probability p_c of bit errors due to carry bits arriving too late is larger in the least significant bits. Using the PCMOS voltage scaling approach, we also employed a probabilistic Wallace-tree multiplier (PWTM) component and added a related energy model and instructions enabling the use of the probabilistic components to our ARMv4 architecture simulator.

We investigated whether FEHLER reliability annotations would also be applicable to determine which arithmetic operations of a program could be executed on PCMOS-based arithmetic components instead of a less energy-efficient traditional ALU without sacrificing the program's stability [10]. A first evaluation using floating point data objects showed that the use of PCMOS technology has the potential for significant energy conservation. Accordingly, we investigated the possible conservation potential for a real-world embedded application. FEHLER type qualifiers were used to indicate data which accepts precision deviation (`unreliable`). Accordingly, our compiler backend generated instructions using probabilistic arithmetic instructions operating on these data objects.

Table 4 shows that a significant fraction of arithmetic ARM machine instructions of our H.264 video decoder could be executed safely on probabilistic components.[6]

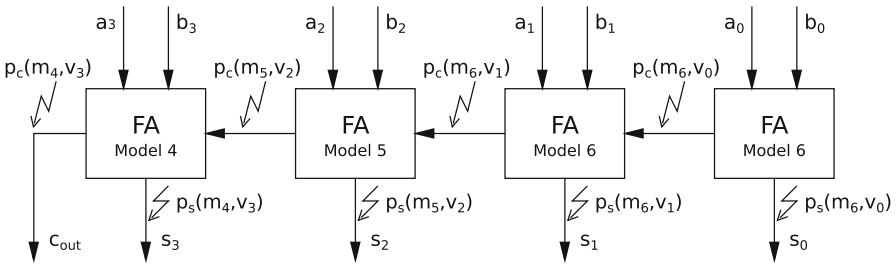


Fig. 9 Probabilistic ripple-carry adder

[6] `rsb` is the ARM reverse subtract instruction.

Table 4 Instructions executed using probabilistic components

Instruction type	Add	Sub	rsb	Mul	Overall
Executed using PRCA/PWTM	18.59%	18.60%	43.01%	76.27%	13.36%

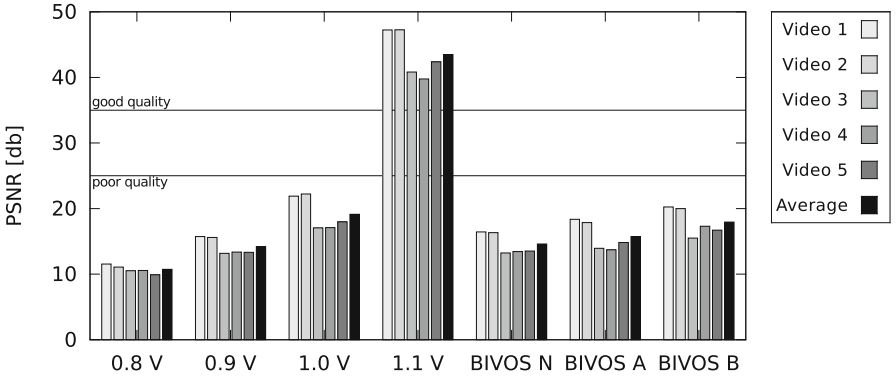


Fig. 10 PSNR (peak signal-to-noise ratio) values for different supply voltage configurations

Surprisingly, our results indicated that for our typical embedded H.264 decoder application, the use of PCMOS components did not result in energy conservation for the identical level of QoS compared to uniform voltage scaling.[7] This result contradicts the microbenchmarks described in [4]. Figure 10 shows the PSNR of the H.264 decoder output for different video clips decoded with circuits using four different UVOS (0.8 V–1.1 V) as well as three BIVOS schemes with similar energy consumption to the UVOS schemes. It can be observed that the PSNR of the BIVOS-decoded videos does not increase, which is a counterintuitive result at first.

A subsequent investigation of the differences between our H.264 decoder and the code used in the microbenchmarks gave insights into the observed effects. Whereas the microbenchmarks employed floating point numbers, our video decoder is a typical embedded application that employs integer and fixed-point numbers.

```
void enter(unreliable uchar *ptr, unreliable int q_delta) {
    unreliable int i = *ptr + ((q_delta + 32) >> 6);
    *ptr=Clip(i);
}
```

Listing 4 H.264 decoder clipping code

This difference in data representation is one of the reasons for the observed phenomenon. The H.264 specification requires a special behavior when copying 32 bit integer values into an eight bit value in the frame buffer. Here, a saturating clipping function (cf. Listing 4) is used. This function restricts the value to 255 if

[7] This only concerns the static and dynamic energy consumption of the PCMOS components. The additional static energy required by the traditional ALU has not been considered here.

the input is larger than that. Accordingly, the shift operation used has the ability to eliminate bit errors in the least significant bits, diminishing the gains of BIVOS scaling.

In contrast, floating point values are always normalized after arithmetic operations. This implies that the bits most relevant to a floating point number's value–sign, exponent, and the MSBs of the mantissa–are always the MSBs of the memory word. In this case, the BIVOS approach to construct arithmetic components that show larger error probabilities in the LSBs is beneficial.

Since it is unrealistic to assume that separate adders for different data widths and data types will be provided in future architectures, an analysis of the number of bits actually used in arithmetic operations is required. However, this implies further complications. One idea for future compiler-based analyses is an approach that combines bit-width analysis methods for arithmetic operations and code transformations to use bits with optimal supply voltage for the operation at hand. The effectiveness of this approach, however, requires further implementation and analysis work.

10 Summary and Outlook

The results of the FEHLER project have shown that for a large class of embedded applications, software-based fault tolerance is a feasible way to reduce the overhead of error handling. The results, as demonstrated using real-world applications, show that already the simple binary classification employed so far is able to avoid crashes due to soft errors while reducing the size of the reliable computing base, i.e. the amount and size of hard- and software components requiring protection from errors.

The technologies developed in the context of FEHLER suggest a number of ways to further improve on the ideas and design of the approach. One constraint of the current design is that the current version of reliability type qualifiers is too coarse-grained. Correcting only errors that affect `reliable` data objects will result in avoiding program crashes. However, a sufficiently high error rate affecting `unreliable` data might still result in a significant reduction of the QoS, rendering its output useless.

The existing static analysis in FEHLER is based on subtyping. Accordingly, to provide a more fine-grained classification of errors, additional error classes have to be introduced. These classes would have to be characterized according to a given total order, so that an error can be classified with the correct worst-case effect. If, for example, the impact of errors is measured in the degradation of a signal-to-noise ratio, a total order can be determined by the resulting amount of degradation.

However, for the overall assessment of a program's QoS, the resulting overall error visible in the output that accumulated throughout the data flow is relevant. Here, one can imagine setting an acceptable QoS limit for the output data and backtracking throughout the arithmetical operations in the program's data flow to

determine the *worst-case deviation* that an error in a given variable can cause in the output. Here, we intend to employ approaches related to numerical error propagation analysis.

We expect that approximate computing approaches will be able to directly benefit from these analyses. Since the approximations already trade precision for other non-functional properties, such as energy consumption, a Pareto optimization of the differing objectives could benefit from worst-case QoS deviation analyses. Here, our initial analysis of the use of binary classifiers for the PCMOS case has already given some interesting preliminary insights.

References

1. Austin, T., Bertacco, V., Mahlke, S., Cao, Y.: Reliable systems on unreliable fabrics. IEEE Des. Test Comput. **25**(4), 322–332 (2008)
2. Bund, T., Slomka, F.: Sensitivity analysis of dropped samples for performance-oriented controller design. In: 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, pp. 244–251 (2015)
3. Chakrapani, L.N., Akgul, B.E.S., Cheemalavagu, S., Korkmaz, P., Palem, K.V., Seshasayee, B.: Ultra-efficient (embedded) SoC architectures based on probabilistic CMOS (PCMOS) technology. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06), pp. 1110–1115. European Design and Automation Association (2006)
4. Chakrapani, L.N., Muntimadugu, K.K., Lingamneni, A., George, J., Palem, K.V.: Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation. In: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08), pp. 187–196. ACM, New York (2008)
5. Chen, K.H., Bönninghoff, B., Chen, J.J., Marwedel, P.: Compensate or ignore? Meeting control robustness requirements through adaptive soft-error handling. In: Languages, Compilers, Tools and Theory for Embedded Systems (LCTES). ACM, Santa Barbara (2016)
6. Engel, M., Döbel, B.: The reliable computing base—a paradigm for software-based reliability. In: INFORMATIK 2012, pp. 480–493. Gesellschaft für Informatik e.V., Bonn (2012)
7. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99), pp. 192–203. ACM, New York (1999)
8. Heinig, A., Engel, M., Schmoll, F., Marwedel, P.: Improving transient memory fault resilience of an H.264 decoder. In: Proceedings of the Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2010). IEEE Computer Society Press, Scottsdale (2010)
9. Heinig, A., Engel, M., Schmoll, F., Marwedel, P.: Using application knowledge to improve embedded systems dependability. In: Proceedings of the Workshop on Hot Topics in System Dependability (HotDep 2010). USENIX Association, Vancouver (2010)
10. 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)
11. Heinig, A., Schmoll, F., Bönninghoff, B., Marwedel, P., Engel, M.: Fame: flexible real-time aware error correction by combining application knowledge and run-time information. In: Proceedings of the 11th Workshop on Silicon Errors in Logic-System Effects (SELSE) (2015)
12. Heinig, A., Schmoll, F., Marwedel, P., Engel, M.: Who's using that memory? A subscriber model for mapping errors to tasks. In: Proceedings of the 10th Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA, USA (2014)

13. 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)
14. Ramanathan, P.: Overload management in real-time control applications using (m, k)-firm guarantee. IEEE Trans. Parallel Distrib. Syst. **10**(6), 549–559 (1999)
15. 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)
16. Whitaker, A., Shaw, M., Gribble, S.: Denali: lightweight virtual machines for distributed and networked applications. In: Proceedings of the 2002 USENIX Annual Technical Conference (2002)
17. Yayla, M., Chen, K., Chen, J.: Fault tolerance on control applications: empirical investigations of impacts from incorrect calculations. In: 2018 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), pp. 17–24 (2018)

ASTEROID and the Replica-Aware Co-scheduling for Mixed-Criticality

Eberle A. Rambo and Rolf Ernst

1 The ASTEROID Project

1.1 Motivation

Technology downscaling has increased the hardware's overall susceptibility to errors to the point where they became non-negligible [17, 21, 22]. Hence, current and future computing systems must be appropriately designed to cope with errors in order to provide a reliable service and correct functionality [17, 21]. That is a challenge, especially in the real-time mixed-criticality domain where applications with different requirements and criticalities co-exist in the system, which must provide *sufficient independence* and prevent error propagation (e.g., timing, data corruption) between criticalities [24, 42]. Recent examples are increasingly complex applications such as flight management systems (FMS), advanced driver assistance systems (ADAS), and autonomous driving (AD) in the avionics and automotive domains, respectively [24, 42]. A major threat to the reliability of such systems is the so-called soft errors.

Soft errors, more specifically Single Event Effects (SEEs), are transient faults abstracted as *bit-flips* in hardware and can be caused by alpha particles, energetic neutrons from cosmic radiation, and process variability [15, 22]. Soft errors are comprehensively discussed in chapter "Reliable CPS Design for Unreliable Hardware Platforms". Depending on where and when they occur, their impact on software execution range from masked (no observable effect) to a complete system crash [3, 12, 13]. Soft errors are typically more frequent than hard errors

E. A. Rambo (✉) · R. Ernst
Institute of Computer and Network Engineering (IDA), TU Braunschweig, Braunschweig, Germany
e-mail: rambo@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_3

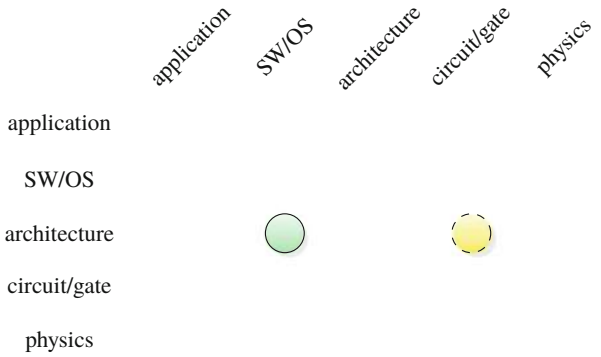


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

(permanent faults) and often remain undetected, also known as latent error or silent data corruption, because they cannot be detected by testing. Moreover, undetected errors are a frequent source of later system crashes [12]. To handle soft errors, the approaches can vary from completely software-based to completely hardware-based. The former are able to cover only part of the errors [12, 13] and the latter result in costly redundant hardware [22], as seen in lock-step dual-core execution [29]. Cross-layer solutions can be more effective and efficient by distributing the tasks of detecting errors, handling them and recovering from them in different layers of software and hardware [12, 13, 22].

1.2 Overview

The ASTEROID project [5] developed a cross-layer fault-tolerance solution to provide reliable software execution on unreliable hardware. The approach is based on replicated software execution and exploits the large number of cores available in modern and future architectures at a higher level of abstraction without resorting to hardware redundancy [5, 12]. That concentrates ASTEROID's contributions around the architecture and operating system (OS) abstraction layers, as illustrated in Fig. 1. ASTEROID's architecture is illustrated in Fig. 2. The reliable software execution is realized by the OS service Romain [12]. Mixed-critical applications may co-exist in the system and are translated into protected and unprotected applications. Romain replicates the protected applications, which are mapped to arbitrary cores, and manages their execution. Error detection is realized by a set of mechanisms whose main feature is the hardware assisted state comparison, which compares the replicas' state at certain points in time [5, 12]. Error recovery strategies can vary depending on whether the application is running in dual modular redundancy (DMR) or triple modular redundancy (TMR) [3, 5].

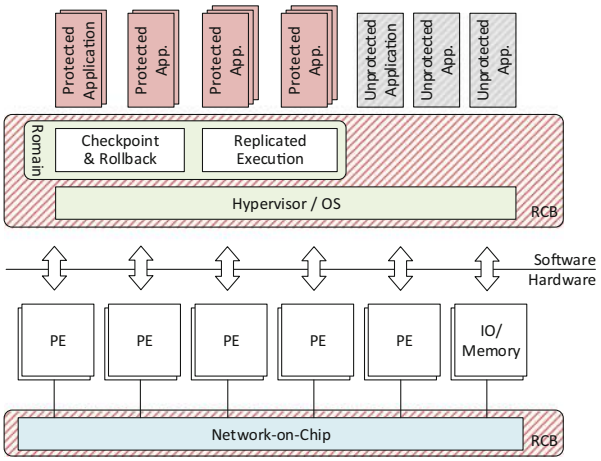


Fig. 2 ASTEROID's architecture

ASTEROID comprised topics ranging from system-level conceptual modeling, to the OS and all the way down to Application-Specific Integrated Circuit (ASIC) synthesis and gate-level simulation. We summarize selected work that were developed in the project next. An initial overview of the ASTEROID approach was introduced in [5]. Romain, the OS service that provides replicated execution for unmodified binary applications, was introduced in [12]. The vulnerabilities of the system were assessed in [9, 13], giving rise to the reliable computing base (RCB), the minimum set of software and hardware components on which the approach relies. The runtime overheads related with the OS-assisted replication were investigated in [10]. Later, RomainMT extends Romain in [11] to support unmodified multithreaded applications. A systematic design process was investigated in [28], followed by the definition of a trusted component ecosystem in [19].

In terms of modeling, the reliability of replicated execution was modeled and evaluated in [3]. The approach was modeled in Compositional Performance Analysis (CPA), a worst-case performance analysis framework, as fork-join tasks and the performance evaluated in [6] and revised in [2]. Later, co-scheduling was employed to improve the worst-case performance of replicated execution with the replica-aware co-scheduling for mixed-criticality [34]. Off-chip real-time communication under soft errors was modeled in [4] with a probabilistic response-time analysis. On-chip real-time communication with and without soft errors were modeled in CPA and evaluated in [33] and [38], where Automatic Repeat reQuest (ARQ)-based protocols were employed in a real-time Network-on-Chip (NoC). As part of the RCB, the NoC's behavior under soft errors was further researched with thorough Failure Mode and Effects Analyses (FMEAs) in [35–37]. Based on those findings, a resilient NoC architecture was proposed in [32, 39, 40], which is able to provide a reliable and predictable service under soft errors.

The remainder of this chapter focuses on the performance of replicated execution under real-time constraints, first published in [34]. It is organized as follows. Section 2 introduces the replica-aware co-scheduling for mixed-criticality and its related work. Section 3 describes the system, task, and error models. Section 4 introduces the formal response-time analysis. Experimental results are reported in Sect. 5. Finally, Sect. 6 ends the chapter with the conclusions.

2 Replica-Aware Co-scheduling for Mixed-Criticality

2.1 Motivation

Replicated software execution is a flexible and powerful mechanism to increase the reliability of the software execution on unreliable hardware. However, the scheduler has a direct influence on its performance. The performance of replicated execution for real-time applications has been formally analyzed in [6] and revised in [2]. The work considers the well-known Partitioned Strict Priority Preemptive (SPP) scheduling, where tasks are mapped to arbitrary cores, and assumes a single error model. The authors found that SPP, although widely employed in real-time systems, provides very pessimistic response-time bounds for replicated tasks. Depending on the interfering workload, replicated tasks executing serially (on the same core) present much better performance than when executing in parallel (on distinct cores). That occurs due to the long time that replicated tasks potentially have to wait on each core to synchronize and compare states before resuming execution. That leads to very low resource utilization and prevents the use of replicated execution in practice.

The replica-aware co-scheduling for mixed-criticality explores co-scheduling to provide short response times for replicated tasks without hindering the remaining unprotected tasks. Co-scheduling is a technique that schedules interacting tasks/threads to execute simultaneously on different cores [30]. It allows tasks/threads to communicate more efficiently by reducing the time they are blocked during synchronization. In contrast to SPP [2, 6], the proposed replica-aware co-scheduling approach drastically minimizes delays due to the implicit synchronization found in state comparisons. In contrast to gang scheduling [14], it rules out starvation and distributes the execution of replicas in time to achieve short response times of unprotected tasks. The proposed approach differs from standard Time-Division Multiplexing (TDM) and TDM with background partition [25] in that all tasks have formal guarantees. In contrast to related work, it supports different recovery strategies and accounts for the NoC communication delay and overheads due to replica management and state comparison. Experimental results with benchmark applications show an improvement on taskset schedulability of up to 6.9× when compared to SPP, and 1.5× when compared to a TDM-based scheduler.

2.2 Related Work

L4/Romain [12] is a cross-layer fault-tolerance approach that provides reliable software execution under soft errors. Romain provides protection at the application-level by replicating and managing the applications' executions as an operating system service. The error detection is realized by a set of mechanisms [5, 12, 13] whose main feature is the hardware assisted state comparison, which allows an effective and efficient comparison of the replicas' states. Pipeline fingerprinting [5] provides a checksum of the retired instructions and the pipeline's data path in every processor, detecting errors in the execution flow and data. The state comparison, reduced to comparing checksums instead of data structures, is carried out at certain points in time. It must occur at least when the application is about to externalize its state, e.g., in a *syscall* [12]. The replica generated *syscalls* are intercepted by Romain, have their integrity checked, and their replicas' states compared before being allowed to externalize the state [12].

Mixed-criticality, in the context of the approach, is supported with different levels of protection for applications with different criticalities and requirements (unprotected, protected with DMR[1] or TMR) and by ensuring that timing constraints are met even in case of errors. For instance, Romain provides different error recovery strategies [3, 5]:

- *DMR with checkpoint and rollback*: to recover, the replicas rollback to their last valid state and re-execute;
- *TMR with state copy*: to recover, the state of the faulty replica is replaced with the state of one of the healthy replicas.

This chapter focuses on the system-level timing aspect of errors affecting the applications. We assume thereby the absence of failures in critical components [13, 32], such as the OS/hypervisor, the replica manager/voter (e.g., Romain), and interconnect (e.g., NoC), which can be protected as in [23, 39].

The Worst-Case Response Time (WCRT) of replicated execution has been analyzed in [6], where replicas are modeled as fork-join tasks in a system implementing Partitioned SPP. The work was later revised in [2] due to optimism in the original approach. The revised approach is used in this work. In that approach, with deadline monotonic priority assignment, where the priority of tasks decreases as their deadlines increase, replicated tasks perform worse when mapped in parallel than when mapped to the same core. This is due to the state comparisons during execution, which involves implicit synchronization between cores. With partitioned scheduling, in the worst-case, the synchronization ends up accumulating the interference from all cores to which the replicated task is mapped, resulting in poor performance at higher loads. On the other hand, mapping replicated tasks

[1] DMR *per se* can be used for system integrity only. However, DMR augmented with checkpointing and rollback enables recovery and can be used to achieve integrity and availability (state rollback followed by re-execution in both replicas) [3, 5].

to the highest priorities results in long response times for lower priority tasks and rules out deadline monotonicity. The latter causes the unschedulability of all tasksets with at least one regular task whose deadline is shorter than the execution time of a replicated task.

Gang scheduling [14] is a co-scheduling variant that schedules groups of interacting tasks/threads simultaneously. It increases performance by reducing the inter-thread communication latency. The authors in [26] present an integration between gang scheduling and Global Earliest Deadline First (EDF), called the Gang EDF. They provide a schedulability analysis derived from the Global EDF's based on the sporadic task model. In another work, [16] shows that SPP Gang schedulers in general are not predictable, for instance, due to priority inversions and slack utilization. In the context of real-time systems, gang scheduling has not received much attention.

TDM-based scheduling [25] is widely employed to achieve predictability and ensure temporal-isolation. Tasks are allocated to partitions, which are scheduled to execute in time slots. Partitions can span across several (or all) cores and can be executed at the same time. The downside of TDM is that it is not work-conserving and underutilizes system resources. A TDM variant with background partition [25] tackles this issue by allowing low priority tasks to execute in other partitions whenever no higher priority workload is executing. Yet, in addition to the high cost to switch between partitions, no guarantees can be given to tasks in the background partition.

In the proposed approach, we exploit co-scheduling with SPP to improve the performance of the system. The proposed approach differs from [6] in that replicas are treated as gangs and are mapped with highest priorities, and are hence activated simultaneously on different cores. In contrast to gang scheduling [14, 16] and to [6], the execution of replicas is distributed in time with offsets to compensate for the lack of deadline monotonicity, thus allowing the schedulability of tasks with short deadlines. We further provide for the worst-case performance of lower priority tasks by allowing them to execute whenever no higher priority workload is executing. However, in contrast to [25], all tasks have WCRT guarantees. Moreover, we also model the state comparison and the on-chip communication overheads.

3 System, Task, and Error Models

In this work, we use the CPA [20] to provide formal response-time bounds. Let us introduce the system, task, and error models.

3.1 System Model

The system consists of a standard NoC-based many-core composed of processing elements, simply referred to as cores.

There are two types of tasks in our system, as in [2]:

- *independent* tasks τ_i: regular, unprotected tasks; and
- *fork-join* tasks Γ_i: replicated, protected tasks.

The system implements partitioned scheduling, where the operating system manages tasks statically mapped to cores. The mapping is assumed to be given as input. The scheduling policy is a combination of SPP and gang scheduling. When executing only independent tasks, the system's behavior is identical to Partitioned SPP, where tasks are scheduled independently on each core according to SPP. It differs from SPP when scheduling fork-join tasks.

Fork-join tasks are mapped with highest priorities, hence do not suffer interference from independent tasks, and execute simultaneously on different cores, as in gang scheduling. Note that deadline monotonicity is, therefore, only partially possible. To limit the interference to independent tasks, the execution of a fork-join task is divided in smaller intervals called stages, whose executions are distributed in time. At the end of each stage, the states of the replicas are compared. In case of an error, i.e. states differ, recovery is triggered.

Fork-join stages are executed with static offsets [31] in execution slots. One stage is executed per slot. On a core with n fork-join tasks, there are $n + 1$ execution slots: one slot for each fork-join task Γ_i and one slot for recovery. The slots are cyclically scheduled in a cycle Φ. The slot for Γ_i starts at offset $\phi(\Gamma_i)$ relative to the start of Φ and ends after $\varphi(\Gamma_i)$, the slot length. The recovery slot is shared by all fork-join tasks on that core and is where error recovery may take place under a single error assumption (details in Sects. 3.3 and 4.3). The recovery slot has an offset $\phi(recovery)$ relative to Φ and length $\varphi(recovery)$. Lower priority independent tasks are allowed to execute whenever no higher priority workload is executing.

An example is shown in Fig. 3, where two fork-join tasks Γ_1 and Γ_2 and two independent tasks τ_3 and τ_4 are mapped to two cores. Γ_1 and Γ_2 execute in their respective slots simultaneously in both cores. When an error occurs, the recovery of Γ_2 is scheduled and the recovery of the error-affected stage occurs in the recovery slot. The use of offsets enables the schedulability of independent tasks with short periods and deadlines, such as τ_3 and τ_4. Note that, without the offsets, Γ_1 and Γ_2 would execute back-to-back leading to the unschedulability of τ_3 and τ_4.

3.2 Task Model

An independent task τ_i is mapped to core σ with a priority p. Once activated, it executes for at most C_i, its worst-case execution time (WCET). The activations of

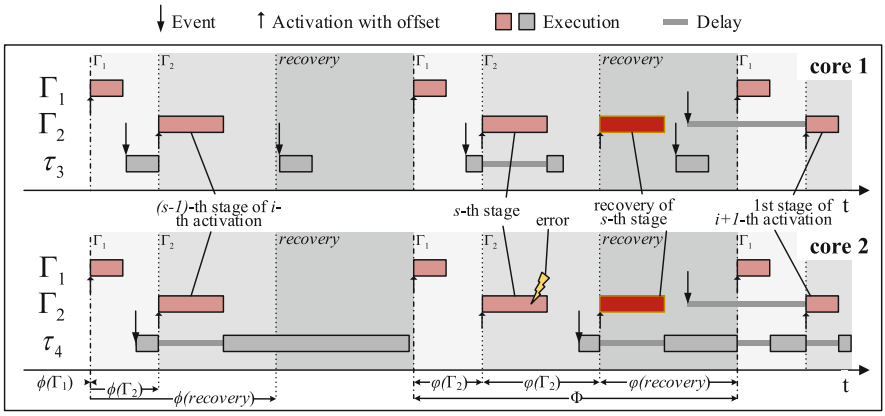


Fig. 3 Execution example with two fork-join and two independent tasks on two cores [34]

a task are modeled with arbitrary *event models*. Task activations in an event model are given by arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, which return the minimum and maximum number of events arriving in any time interval Δt. Their pseudo-inverse counterparts $\delta^+(q)$ and $\delta^-(q)$ return the maximum and minimum time interval between the first and last events in any sequence of q event arrivals. Conversion is provided in [41]. Periodic events with jitter, sporadic events, and others can be modeled with the minimum distance function $\delta_i^-(q)$ as follows [41]:

$$\delta_i^-(q) = \max((q-1)\cdot d^{min}, (q-1)\cdot \mathcal{P} - \mathcal{J}) \tag{1}$$

where $\mathcal{P}$ is the period, $\mathcal{J}$ is the jitter, d^{min} is the minimum distance between any two events, and the subscript i indicates the association with a task τ_i or Γ_i.

Fork-join tasks are rigid parallel tasks, i.e. the number of processors required by a fork-join task is fixed and specified externally to the scheduler [16], and consist of multiple stages with data dependencies, as in [1, 2]. A fork-join task Γ_i is a Directed Acyclic Graph (DAG) $G(V, E)$, where vertices in V are subtasks and edges in E are precedence dependencies [2]. In the graph, tasks are partitioned in *segments* and *stages*, as illustrated in Fig. 5a. A subtask $\tau_i^{\sigma,s}$ is the s-th stage of the σ-th segment and is annotated with its WCET $C_i^{\sigma,s}$. The WCET of a stage is equal across all segments, i.e. $\forall x, y : C_i^{x,s} = C_i^{y,s}$. Each segment σ of Γ_i is mapped to a distinct core. A fork-join task Γ_i is annotated with the *static offset* $\phi(\Gamma_i)$, which marks the start of its execution slot in Φ. The offset also admits a small positive jitter j_ϕ, to account for a slight desynchronization between cores and context switch overhead.

The activations of a fork-join task are modeled with *event models*. Once Γ_i is activated, its stages are successively activated by the completion of all segments of the previous stage, as in [1, 2]. Our approach differs from them in that it restricts the scheduling of at most one stage of Γ_i in a cycle Φ, and the stage receives service at the offset $\phi(\Gamma_i)$. Note that the event arrival at a fork-join task is not synchronized

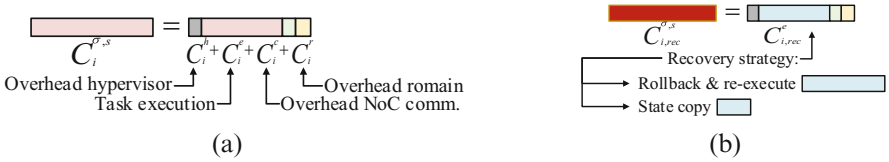


Fig. 4 The composition of WCET of fork-join subtasks [34]. (**a**) WCET of a fork-join subtask. (**b**) WCET of recovery

with its offset. The events at a fork-join task are queued at the first stage and only one event at a time is processed (FIFO) [2]. A queued event is admitted when the previous event leaves the last stage.

The interaction with Romain (the voter) is modeled in the analysis as part of the WCET $C_i^{\sigma,s}$, as depicted in Fig. 4a. The WCET includes the on-chip communication latency and state comparison overheads, as the Romain instance may be mapped to an arbitrary core. Those can be obtained, e.g., with [38] along with task mapping and scheduler properties to avoid over-conservative interference estimation and obtain tighter bounds.

3.3 Error Model

Our model assumes a single error scenario caused by SEEs. We assume that all errors affecting fork-join tasks can be detected and contained, ensuring integrity. The overhead of error detection mechanisms is modeled as part of the WCET (cf. Fig. 4a). Regarding independent tasks, we assume that an error immediately leads to a task failure and assume also that its failure will not violate the WCRT guarantees of the remaining tasks. Those assumptions are met, e.g., by Romain.[2] Moreover, we assume the absence of failures in critical components [13, 32], such as the OS, the replica manager/voter Romain, and the interconnect (e.g., the NoC), which can be protected as in [23, 39].

Our model provides recovery[2] for fork-join tasks, ensuring their availability. With a recovery slot in every cycle Φ, our approach is able to handle up to one error per cycle Φ. However, the analysis in Sect. 4.3 assumes at most one error per busy window for the sake of a simpler analysis (the concept will be introduced in Sect. 4). The assumption is reasonable since the probability of a multiple error scenario is very low and can be considered as an acceptable risk [24]. A multiple error scenario occurs only if an error affects more than one replica at a time or if more than one error occurs within the same busy window.

[2]Romain is able to detect and recover from all soft errors affecting user-level applications. For details on the different error impacts and detection strategies, the interested reader can refer to [5, 12].

3.4 Offsets

The execution of fork-join tasks in our approach is based on static offsets, which are assumed to be provided as input to the scheduler. The offsets form execution slots whose size do not vary during runtime, as seen in Fig. 3. Varying the slots sizes would substantially increase the timing analysis complexity without a justifiable performance gain. The offsets must satisfy two constraints:

Constraint 1 *A slot for a fork-join task Γ_i must be large enough to fit the largest stage of Γ_i. That is, $\forall s, \sigma$: $\varphi(\Gamma_i) \geq C_i^{\sigma,s} + j_\phi$.*

Constraint 2 *The recovery slot must be large enough to fit the recovery of the largest stage of any fork-join task mapped to that core. That is, $\forall i, s, \sigma$: $\varphi(recovery) \geq C_{i,rec}^{\sigma,s} + j_\phi$.*

where a one error scenario per cycle is assumed and $C_{i,rec}^{\sigma,s}$ is the recovery WCET of subtask $\tau_i^{\sigma,s}$ (cf. Sect. 4.3).

We provide basic offsets that satisfy Constraints 1 and 2. The calculation must consider only overlapping fork-join tasks, i.e. fork-join tasks mapped to at least one core in common. Offsets for non-overlapping fork-join tasks are computed separately as they do not interfere directly with each other. The indirect interference, e.g., in the NoC, is accounted for in the WCETs. First we determine the smallest slots that satisfy Constraint 1:

$$\forall\, \Gamma_i : \varphi(\Gamma_i) = \max_{\forall \sigma, s} \left\{ C_i^{\sigma,s} \right\} + j_\phi \tag{2}$$

and the smallest recovery slot that satisfies Constraint 2:

$$\varphi(recovery) = \max_{\forall \Gamma_i, \tau_j^{\sigma,s} \in \Gamma_i} \left\{ C_{i,rec}^{\sigma,s} \right\} + j_\phi \tag{3}$$

The cycle Φ is then the sum of all slots:

$$\Phi = \sum_{\forall \Gamma_i} \{\varphi(\Gamma_i)\} + \varphi(recovery) \tag{4}$$

The offsets then depend on the order in which the slots are placed inside Φ. Assuming that the slots $\phi(\Gamma_i)$ are sorted in ascending order on i and that the recovery slot is the last one, the offsets are obtained by

$$\phi(x) = \begin{cases} 0 & \text{if } x = \Gamma_1 \\ \phi(\Gamma_{i-1}) + \varphi(\Gamma_{i-1}) & \text{if } x = \Gamma_i \text{ and } i > 1 \\ \Phi - \varphi(recovery) & \text{if } x = recovery \end{cases} \tag{5}$$

4 Response-Time Analysis

The analysis is based on CPA and inspired by Axer [2] and Palencia and Harbour [31]. In CPA, the WCRT is calculated with the busy window approach [43]. The response time of an event of a task τ_i (resp. Γ_i) is the time interval between the event arrival and the completion of its execution. In the busy window approach [43], the event with the WCRT can be found inside the busy window. The busy window w_i of a task τ_i (resp. Γ_i) is the time interval where all response times of the task depend on the execution of at least one previous event in the same busy window, except for the task's first event. The busy window starts at a critical instant corresponding to the worst-case scheduling scenario. Since the worst-case scheduling scenario depends on the type of task, it will be derived individually in the sequel.

Before we derive the analysis for fork-join and for independent tasks, let us introduce the example in Fig. 5 used throughout the section. The taskset consists of four independent tasks and two fork-join tasks, mapped to two cores. The task priority on each core decreases from top to bottom (e.g., $\tau_1^{1,1}$ has the highest priority and τ_4 the lowest).

4.1 Fork-Join Tasks

We now derive the WCRT for an arbitrary fork-join task Γ_i. To do that, we need to identify the critical instant leading to the worst-case scheduling scenario. In case

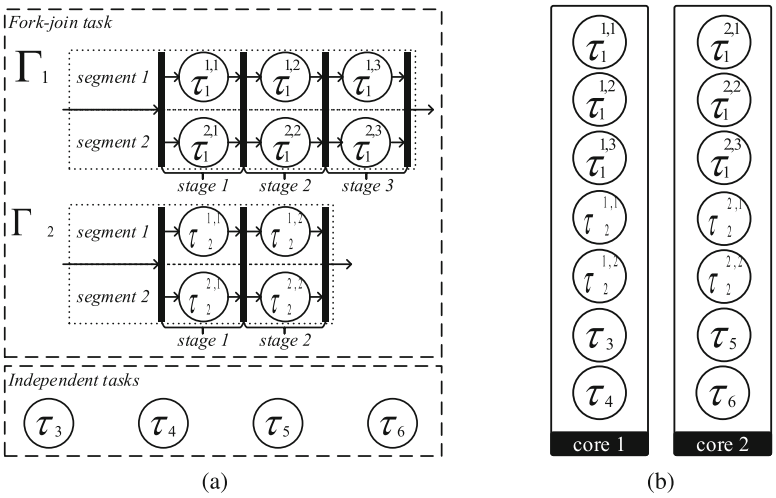


Fig. 5 A taskset with 4 independent tasks and 2 fork-join tasks, and its mapping to 2 cores. Highest priority at the top, lowest at the bottom [34]. (**a**) Taskset. (**b**) Mapping

of SPP, the critical instant is when all tasks are activated at the same time and the tasks' subsequent events arrive as early as possible [43]. In our case, the critical instant must also account for the use of static offsets [31].

The worst-case scheduling scenario for Γ_2 on core 1 is illustrated in Fig. 6. Γ_2 is activated and executed at the same time on cores 1 and 2 (omitted). Note that, by design, fork-join tasks do not dynamically interfere with each other. The *critical instant* occurs when the first event of Γ_2 arrives just after missing Γ_2's offset. The event has to wait until the next cycle to be served, which takes time $\Phi + j_\phi$ when the activation with offset is delayed by a jitter j_ϕ. Notice that the WCETs of fork-join tasks already account for the inter-core communication and synchronization overhead (cf. Fig. 4a).

Lemma 1 *The critical instant leading to the worst-case scheduling scenario of a fork-join task Γ_i is when the first event of Γ_i arrives just after missing Γ_i's offset $\phi(\Gamma_i)$.*

Proof A fork-join task Γ_i does not suffer interference from independent tasks or other fork-join tasks. The former holds since independent tasks always have lower priority. The latter holds due to three reasons: an arbitrary fork-join task Γ_j always receives service in its slot $\phi(\Gamma_j)$; the slot $\phi(\Gamma_j)$ is large enough to fit Γ_j's largest subtask (Constraint 1); and the slots in a cycle Φ are disjoint. Thus, the critical instant can only be influenced by Γ_i itself.

We prove by contradiction. Suppose that there is another scenario worse than Lemma 1. That means that the first event can arrive at a time that causes a delay to Γ_i larger than $\Phi + j_\phi$. However, if the delay is larger than $\Phi + j_\phi$, then the event arrived before a previous slot $\phi(\Gamma_i)$ and Γ_i did not receive service. Since that can only happen if there is a pending activation of Γ_i and thus violates the definition of a busy window, the hypothesis must be rejected. □

Let us now derive the Multiple-Event Queueing Delay $Q_i(q)$ and Multiple-Event Busy Time $B_i(q)$ on which the busy window relies. $Q_i(q)$ is the longest time interval between the arrival of Γ_i's first activation and the first time its q-th activation

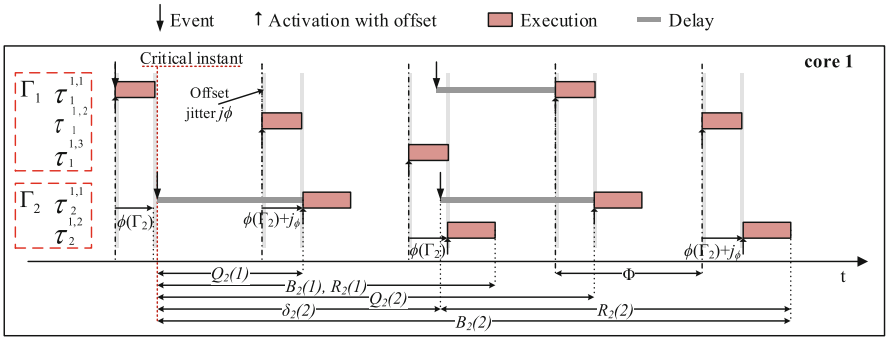


Fig. 6 Worst-case schedule for fork-join gang Γ_2 on core 1 (cf. Fig. 5) [34]

receives service, considering that all events belong to the same busy window [2, 27]. For Γ_i, the q-th activation can receive service at the next cycle Φ after the execution of $q-1$ activations of Γ_i lasting $s_i \cdot \Phi$ each, a delay Φ (cf. (cf. Lemma 1), and a jitter j_ϕ. This is given by

$$Q_i(q) = (q-1) \cdot s_i \cdot \Phi + \Phi + j_\phi \tag{6}$$

where s_i is the number of stages of Γ_i and Φ is the cycle.

Lemma 2 *The Multiple-Event Queueing Delay $Q_i(q)$ given by Eq. 6 is an upper bound.*

Proof The proof is by induction. When $q = 1$, Γ_i has to wait for service at most until the next cycle Φ plus an offset jitter j_ϕ to get service for its first stage, considering that the event arrives just after its offset (Lemma 1). In a subsequent $q + 1$-th activation in the same busy window, Eq. 6 must also consider q entire executions of Γ_i. Since Γ_i has s_i stages and only one stage can be activated and executed per cycle Φ, it takes additional $s_i \cdot \Phi$ for each activation of Γ_i, resulting in Eq. 6. □

The Multiple-Event Busy Time $B_i(q)$ is the longest time interval between the arrival of Γ_i's first activation and the completion of its q-th activation, considering that all events belong to the same busy window [2, 27]. The q-th activation of Γ_i completes after a delay Φ (cf. Lemma 1), a jitter j_ϕ, and the execution of q activations of Γ_i. This is given by

$$B_i(q) = q \cdot s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \tag{7}$$

where $C_i^{\sigma,s}$ is the WCET of Γ_i's last stage.

Lemma 3 *The Multiple-Event Busy Time $B_i(q)$ given by Eq. 7 is an upper bound.*

Proof The proof is by induction. When $q = 1$, Γ_i has to wait for service at most until the next cycle Φ plus an offset jitter j_ϕ to get service for its first stage (Lemma 1), plus the completion of the last stage of the activation lasting $(s_i - 1) \cdot \Phi + C_i^{\sigma,s}$. This is given by

$$\begin{aligned} B_i(1) &= (s_i - 1) \cdot \Phi + \Phi + j_\phi + C_i^{\sigma,s} \\ &= s_i \cdot \Phi + j_\phi + C_i^{\sigma,s} \end{aligned} \tag{8}$$

In a subsequent $q + 1$-th activation in the same busy window, Eq. 7 must consider q additional executions of Γ_i. Since Γ_i has s_i stages and only one stage can be activated and executed per cycle Φ, it takes additional $s_i \cdot \Phi$ for each activation of Γ_i. Thus, Eq. 7. □

Now we can calculate the busy window and WCRT of Γ_i. The busy window w_i of a fork-join task Γ_i is given by

$$w_i = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q) \mid Q_i(q+1) \geq \delta_i^-(q+1)\} \tag{9}$$

Lemma 4 *The busy window is upper bounded by Eq. 9.*

Proof The proof is by contradiction. Suppose there is a busy window $\breve{w}_i$ longer than w_i. In that case, $\breve{w}_i$ must contain at least one activation more than w_i, i.e. $\breve{q} \geq q+1$. From Eq. 9, we have that $Q_i(\breve{q}) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. □

The response time $R_i(q)$ of the q-th activation of Γ_i in the busy window is given by

$$R_i(q) = B_i(q) - \delta_i^-(q) \tag{10}$$

The worst-case response time R_i^+ is the longest response time of any activation of Γ_i observed in the busy window.

$$R_i^+ = \max_{1 \leq q \leq \eta_i^+(w_i)} R_i(q) \tag{11}$$

Theorem 1 *R_i^+ (Eq. 11) provides an upper bound on the worst-case response time of an arbitrary fork-join task Γ_i.*

Proof The WCRT of a fork-join task Γ_i is obtained with the busy window approach [43]. It remains to prove that the critical instant leads to the worst-case scheduling scenario, that the interference captured in Eqs. 6 and 7 are upper bounds, and that the busy window is correctly captured by Eq. 9. These are proved in Lemmas 1, 2, 3, and 4, respectively. □

4.2 Independent Tasks

We now derive the WCRT analysis of an arbitrary independent task τ_i. Two types of interference affect independent tasks: interference caused by higher priority independent tasks and by fork-join tasks. Let us first identify the critical instant leading to the worst-case scheduling scenario where τ_i suffers the most interference.

Lemma 5 *The critical instant of τ_i is when the first event of higher priority independent tasks arrives simultaneously with τ_i's event at the offset of a fork-join task.*

Proof The worst-case interference caused by a higher priority (independent) task τ_j under SPP is when its first event arrives simultaneously with τ_i's and continue arriving as early as possible [43].

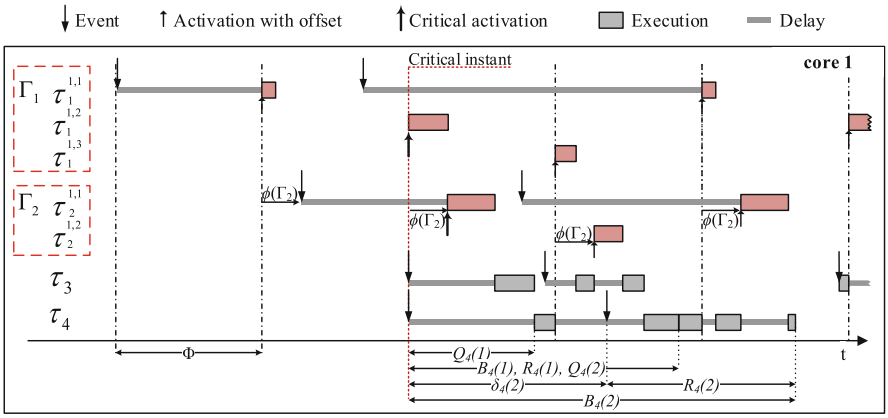


Fig. 7 The worst-case schedule for independent task τ_4 on core 1 (cf. Fig. 5) [34]

The interference caused by a fork-join task Γ_j on τ_i depends on Γ_j's offset $\phi(\Gamma_j)$ and subtasks $\tau_j^{\sigma,s}$, whose execution times vary for different stages s. Assume a critical instant that occurs at a time other than at the offset $\phi(\Gamma_j)$. Since a task Γ_j starts receiving service at its offset, an event of τ_i arriving at time $t > \phi(\Gamma_j)$ can only suffer less interference from Γ_j's subtask than when arriving at $t = 0$. □

Fork-join subtasks have different execution times for different stages, which leads to a number of scheduling scenarios that must be evaluated [31]. Each scenario is defined by the fork-join subtasks that will receive service in the cycle Φ and the offset at which the critical instant supposedly occurs. The scenario is called a critical instant candidate S. Since independent tasks participate in all critical instant candidates, they are omitted in S for the sake of simplicity.

Definition 1 Critical Instant Candidate S: the critical instant candidate S is an ordered pair (a, b), where a is a critical offset and b is a tuple containing one subtask $\tau_j^{\sigma,s}$ of every interfering fork-join task Γ_j.

Let us also define the set of candidates that must be evaluated.

Definition 2 Critical Instant Candidate Set $\mathcal{S}$: the set containing all possible different critical instant candidates S.

The worst-case schedule of the independent task τ_4 from the example in Fig. 5 is illustrated in Fig. 7. In fact, the critical instant leading to τ_4's WCRT is at $\phi(\Gamma_1)$ when $\tau_1^{1,2}$ and $\tau_2^{1,1}$ receive service at the same cycle Φ, i.e. $S = (\phi(\Gamma_1), (\tau_1^{1,2}, \tau_2^{1,1}))$. Events of the independent task τ_3 start arriving at the critical instant and continue arriving as early as possible.

Let us now bound the interference $I_i^I(\Delta t)$ caused by equal or higher priority independent tasks in any time interval Δt. The interference $I_i^I(\Delta t)$ can be upper bounded as follows [27]:

$$I_i^I(\Delta t) = \sum_{\forall \tau_j \in hp_I(i)} \eta_j^+(\Delta t) \cdot C_j \tag{12}$$

where $hp_I(i)$ is the set of equal or higher priority independent tasks mapped to the same core as τ_i.

To derive the interference caused by fork-join tasks we need to define the Critical Instant Event Model. The critical instant event model $\check{\eta}_i^{\sigma,s}(\Delta t, S)$ of a subtask $\tau_i^{\sigma,s} \in \Gamma_i$ returns the maximum number of activations observable in any time interval Δt, assuming the critical instant S. It can be derived from Γ_i's input event model $\eta_i^+(\Delta t)$ as follows:

$$\check{\eta}_i^{\sigma,s}(\Delta t, S) = \min\left\{\eta_i^+\left(\Delta t_S + \Phi - \phi(\Gamma_i)\right), \psi\right\} - gt\left(s^S, s, \phi^S, \phi(\Gamma_i)\right) \tag{13}$$

$$\psi = \left\lfloor \frac{\Delta t_S}{\Phi \cdot s_i} \right\rfloor + ge\left(\Delta t_S \bmod (\Phi \cdot s_i),\ \Phi \cdot (s-1)\right) \tag{14}$$

$$\Delta t_S = \Delta t + \underbrace{\Phi \cdot (s^S - 1)}_{\substack{\text{critical instant} \\ \text{stage}}} + \underbrace{\phi^S}_{\substack{\text{critical instant} \\ \text{offset}}} \tag{15}$$

where s is the stage of subtask $\tau_i^{\sigma,s}$; s_i is the number of stages in Γ_i; ϕ^S is the offset in S; s^S is the stage of Γ_i in S; $gt(a, b, c, d)$ is a function that returns 1 when $(a > b) \vee (a = b \wedge c > d)$, 0 otherwise; and $ge(a, b)$ is a function that returns 1 when $a \geq b$, 0 otherwise.

Lemma 6 *$\check{\eta}_i^{\sigma,s}(\Delta t, S)$ (Eq. 13) provides a valid upper bound on the number of activations of $\tau_i^{\sigma,s}$ observable in any time interval Δt, assuming the critical instant S.*

Proof The proof is by induction, in two parts. First let us assume $s^S = 1$ and $\phi^S = 0$, neutral values resulting in $\Delta t_S = \Delta t$ and $gt(s^S, s, \phi^S, \phi(\Gamma_i)) = 0$. The maximum number of activations of $\tau_i^{\sigma,s}$ seen in the interval Δt is limited by the maximum number of activations of the fork-join task Γ_i because a subtask $\tau_i^{\sigma,s}$ is activated once per Γ_i's activation, and limited by the maximum number of times that $\tau_i^{\sigma,s}$ can actually be scheduled and served in Δt. This is ensured in Eq. 13 by the minimum function and its first and second terms, respectively.

When $s^S > 1$ and/or $\phi^S > 0$, the time interval $[0, \Delta t)$ must be moved forward so that it starts at stage s^S and offset ϕ^S. This is captured by Δt_S in Eq. 15 and by the last term of Eq. 13. The former extends the end of the time interval by the time it takes to reach the stage s^S and the offset ϕ^S, i.e. $[0, \Delta t_S)$. The latter pushes the start of the interval forward by subtracting an activation of $\tau_i^{\sigma,s}$ if it occurs before the stage s^S and the offset ϕ^S, resulting in the interval $[\Delta t_S - \Delta t, \Delta t_S)$. Thus Eq. 13. □

The interference $I_i^{FJ}(\Delta t, S)$ caused by fork-join tasks on the same core in any time interval Δt, assuming a critical instant candidate S, can then be upper bounded

as follows:

$$I_i^{FJ}(\Delta t, S) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{FJ}(i)} \breve{\eta}_j^{\sigma,s}(\Delta t, S) \cdot C_j^{\sigma,s} \tag{16}$$

where $hp_{FJ}(i)$ is the set of fork-join subtasks mapped to the same core as τ_i.

The Multiple-Event Queueing Delay $Q_i(q, S)$ and Multiple-Event Busy Time $B_i(q, S)$ for an independent task τ_i, assuming a critical instant candidate S, can be derived as follows:

$$Q_i(q, S) = (q-1) \cdot C_i + I_i^I(Q_i(q, S)) + I_i^{FJ}(Q_i(q, S), S) \tag{17}$$

$$B_i(q, S) = q \cdot C_i + I_i^I(B_i(q, S)) + I_i^{FJ}(B_i(q, S), S) \tag{18}$$

where $q \cdot C_i$ is the time required to execute q activations of task τ_i.

Equations 17 and 18 result in fixed-point problems, similar to the well-known busy window equation (Eq. 9). They can be solved iteratively, starting with a very small, positive ϵ.

Lemma 7 *The Multiple-Event Queueing Delay $Q_i(q, S)$ given by Eq. 17 is an upper bound, assuming the critical instant S.*

Proof The proof is by induction. When $q = 1$, τ_i has to wait for service until the interfering workload is served. The interfering workload is given by Eqs. 12 and 16. Since $\eta_j^+(\Delta t)$ and C_j are upper bounds by definition, Eq. 12 is also an upper bound. Similarly, since $\breve{\eta}_j^{\sigma,s}(\Delta t, S)$ is an upper bound (cf. Lemma 6) and $C_j^{\sigma,s}$ is an upper bound by definition, 16 is an upper bound for a given S. Therefore, $Q_i(1, S)$ is also an upper bound, for a given S.

In a subsequent $q+1$-th activation in the same busy window, $Q_i(q, S)$ also must consider q executions of τ_i. This is captured in Eq. 17 by the first term, which is, by definition, an upper bound on the execution time. From that, Lemma 7 follows. □

Lemma 8 *The Multiple-Event Busy Time $B_i(q, S)$ given by Eq. 18 is an upper bound, assuming the critical instant S.*

Proof The proof is similar to Lemma 7, except that $B_i(q, S)$ in Eq. 18 also captures the completion of the q-th activation. It takes additional C_i, which is an upper bound by definition. Thus Eq. 18 is an upper bound, for a given S. □

The busy window $w_i(q, S)$ of an independent task τ_i is given by

$$w_i(S) = \max_{q \geq 1,\, q \in \mathbb{N}} \{B_i(q, S) \mid Q_i(q+1, S) \geq \delta_i^-(q+1)\} \tag{19}$$

Lemma 9 *The busy window is upper bounded by Eq. 19.*

Proof The proof is by contradiction. Suppose there is a busy window $\breve{w}_i(S)$ longer than $w_i(S)$. In that case, $\breve{w}_i(S)$ must contain at least one activation more than $w_i(S)$,

i.e. $\breve{q} \geq q + 1$. From Eq. 19, we have that $Q_i(\breve{q}, S) < \delta_i^-(\breve{q})$, i.e. $\breve{q}$ is not delayed by the previous activation. Since that violates the definition of a busy window, the hypothesis must be rejected. □

The response time R_i of the q-th activation of a task in a busy window is given by

$$R_i(q, S) = B_i(q, S) - \delta_i^-(q) \tag{20}$$

Finally, the worst-case response time R_i^+ is found inside the busy window and must be evaluated for all possible critical instant candidates $S \in \mathcal{S}$. The worst-case response time R_i^+ is given by

$$R_i^+ = \max_{S \in \mathcal{S}} \left\{ \max_{1 \leq q \leq \eta_i^+(w_i(S))} \{R_i(q, S)\} \right\} \tag{21}$$

where the set $\mathcal{S}$ is given by the following Cartesian products:

$$\mathcal{S} = \left\{\phi(\Gamma_j), \phi(\Gamma_k), \ldots\right\} \times \left\{\sigma_i(\Gamma_j) \times \sigma_i(\Gamma_k) \times \ldots\right\} \tag{22}$$

where $\Gamma_j, \Gamma_k, \ldots$ are all fork-join tasks mapped to the same core as τ_i and $\sigma_i(\Gamma_j)$ is the set of subtasks of Γ_j that are mapped to that core. When no fork-join tasks interfere with τ_i, the set $\mathcal{S} = \{(0, ())\}$.

Theorem 2 *R_i^+ (Eq. 21) returns an upper bound on the worst-case response time of an independent task τ_i.*

Proof We must first prove that, for a given S, R_i^+ is an upper bound. R_i^+ is obtained with the busy window approach [43]. It returns the maximum response time $R_i(q, S)$ among all activations inside the busy window. From Lemmas 7 and 8 we have that Eqs. 17 and 18 are upper bounds for a given S. From Lemma 9 we have that the busy window is captured by Eq. 19. Since the first term of Eq. 20 is an upper bound and the second term is a lower bound by definition, $R_i(q, S)$ is an upper bound. Thus R_i^+ is an upper bound for a given S. Since Eq. 21 evaluates the maximum response time over all $S \in \mathcal{S}$, R_i^+ is an upper bound on the response time of τ_i. □

4.3 Error Recovery

Designed for mixed-criticality, our approach supports different recovery strategies for different fork-join tasks (cf. Sect. 2.2). For instance, in DMR augmented with checkpointing and rollback, recovery consists in reverting the state and re-executing the error-affected stage in both replicas. In TMR, recovery consists in copying and replacing the state of the faulty replica with the state of a healthy one. The different

strategies are captured in the analysis by the recovery execution time, which depends on the strategy and the stage to be recovered. The recovery WCET $C_{i,rec}^{\sigma,s}$ of a fork-join subtask $\tau_i^{\sigma,s}$ accounts for the adopted recovery strategy as illustrated in Fig. 4b. Once an error is detected, error recovery is triggered and executed in the recovery slot of the same cycle Φ. Figure 3 illustrates the recovery of the s-th stage of Γ_2's i-th activation.

Let us incorporate the error recovery into the analysis. For a fork-join task Γ_i, we must only adapt the Multiple-Event Busy Time $B_i(q)$ (Eq. 7) to account for the execution of the recovery:

$$B_i^{rec}(q) = q \cdot s_i \cdot \Phi + j_\phi + \phi(recovery) - \phi(\Gamma_i) + C_{i,rec}^{\sigma,s} \tag{23}$$

where $C_{i,rec}^{\sigma,s}$ is the WCET of the recovery of last subtask of Γ_i. The recovery of another task Γ_j does not interfere with Γ_i's WCRT. Only the recovery of one of Γ_i's subtasks can interfere with Γ_i's WCRT. Moreover, since the recovery of a subtask occurs in the recovery slot of the same cycle Φ and does not interfere with the next subtask, only the recovery of the last stage of Γ_i actually has an impact on its response time. This is captured by the three last terms of Eq. 23.

For an independent task τ_i, the worst-case impact of recovery of a fork-join task Γ_j is modeled as an additional fork-join task Γ_{rec} with one subtask $\tau_{rec}^{\sigma,1}$ mapped to the same core as τ_i and that executes in the $recovery$ slot. The WCET $C_{rec}^{\sigma,1}$ of $\tau_{rec}^{\sigma,1}$ is chosen as the maximum recovery time among the subtasks of all fork-join tasks mapped to that core:

$$C_{rec}^{\sigma,1} = \max_{\forall \tau_j^{\sigma,s} \in hp_{FJ}(i)} \left\{ C_{i,rec}^{\sigma,s} \right\} \tag{24}$$

with Γ_{rec} mapped, Eq. 21 finds the critical instant, where the recovery $C_{rec}^{\sigma,1}$ has the worst impact on the response time of τ_i.

5 Experimental Evaluation

In our experiments we evaluate our approach with real as well as synthetic workloads, focusing on the performance of the scheduler. First we characterize MiBench applications [18] and evaluate them as fork-join (replicated) tasks in the system. Then we evaluate the performance of independent (regular) tasks. Finally we evaluate the approach with synthetic workloads when varying parameters of fork-join tasks.

Table 1 MiBench applications' profile [34]

	WCET	Observed stages		Grouped stages	
	[ms]	#stages	Max WCET [ms]	#stages	Max WCET [ms]
basicmath	32.48	19,738	0.02	5	6.50
bitcount	24.42	30	15.16	3	15.16
susan	9.63	12	9.59	1	9.63
blowfish	0.11	7	0.09	1	0.11
rijndael	13.17	93	0.37	3	5.91
sha	3.49	51	0.11	2	1.90

5.1 Evaluation with Benchmark Applications

5.1.1 Characterization

First we extract execution times and number of stages from MiBench automotive and security applications [18]. They were executed with small input on an ARMv7@1 GHz and a memory subsystem including a DDR3-1600 DRAM [8]. Table 1 summarizes the total WCET, *observed* number of stages, and WCET of the longest stage (max). A stage is delimited by *syscalls* (cf. Sect. 2.2). We report the observed execution times as WCETs. As pointed out in [2], stages vary in number and execution time depending on the application and on the current activity in that stage (computation/IO). This is seen, e.g., in *susan*, where 99% of the WCET is concentrated in one stage (computation) while the other stages perform mostly IO and are on average 3.34 µs long.

In our approach, the optimum is when all stages of a fork-join task have the same WCET. There are two possibilities to achieve that: to *aligned* very long stages in shorter ones or to *group* short, subsequent stages together. We exploit the latter as it does not require changes to the error detection mechanism or to our model. The results with *grouped* stages are shown on the right-hand side of Table 1. We have first grouped stages without increasing the maximum stage length. The largest improvement is seen in *bitcount*, where the number of stages reduces by one order of magnitude. In cases where all stages are very short, we increase the maximum stage length. When increasing the maximum stage length by two orders of magnitude, the number of stages of *basicmath* reduces by four orders of magnitude. We have manually chosen the maximum stage length. Alternatively the problem of finding the maximum stage length can be formulated as an optimization problem that, e.g., minimizes the overall WCRT or maximizes the slack. Next, we map the applications as fork-join tasks and evaluate their WCRTs.

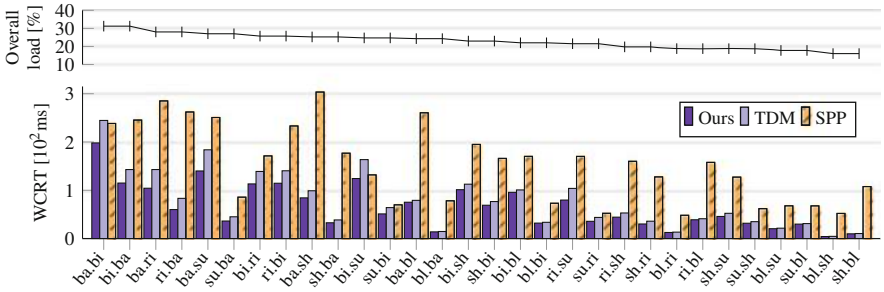


Fig. 8 WCRT of fork-join tasks with two segments derived from MiBench [34]

5.1.2 Evaluation of Fork-Join Tasks

Two applications at a time are mapped as fork-join tasks with two segments (i.e., replicas in DMR) to two cores (cf. Fig. 5). On each core, 15% load is introduced by ten independent tasks generated with UUniFast [7]. We compare our approach with a TDM-based scheduler and Axer's Partitioned SPP [2]. In TDM, each fork-join task executes (and recovers) in its own slot. Independent tasks execute in a third slot, which replaces the recovery slot of our approach. The size of the slots is derived from our offsets. For all approaches, the priority assignment for independent tasks is deadline monotonic and considers that deadline equals period. In SPP, the deadline monotonic priority assignment also includes fork-join tasks.

The results are plotted in Fig. 8, where *ba.bi* gives the WCRT of *basicmath* when mapped together with *bitcount*. Despite the low system load, our approach also outperforms SPP in all cases, with bounds 58.2% lower, on average. Better results with SPP cannot be obtained unless the interfering workload is removed or highest priority is given to the fork-join tasks [2], which violates DM. Despite the similarity of how our approach handles fork-join tasks with TDM, the proposed approach outperforms TDM in all cases, achieving, on average, bounds 13.9% lower. This minor difference is because TDM slots must be slightly longer than our offsets to fit an eventual recovery. Nonetheless, not only our approach can guarantee short WCRT for replicated tasks but also provides for the worst-case performance of independent tasks.

5.1.3 Evaluation of Independent Tasks

In a second experiment we fix *bitcount* and *rijndael* as fork-join tasks and vary the load on both cores. The generated task periods are in the range [20, 500] ms, larger than the longest stage of the fork-join tasks. The schedulability of the system as the load increases is shown in Fig. 9. Our approach outperforms TDM and SPP in all cases, scheduling 1.55× and 6.96× more tasksets, respectively. Due to its non-work conserving characteristic, TDM's schedulability is limited

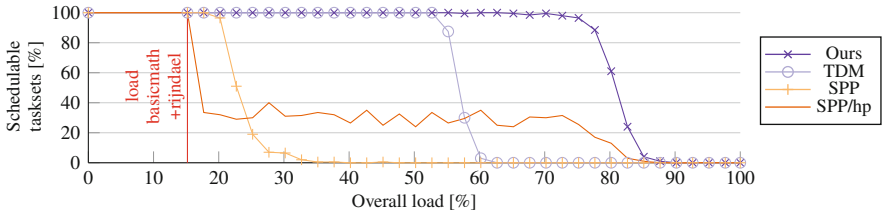


Fig. 9 Schedulability as a function of the load of the system. *Basicmath* and *rijndael* as fork-join tasks with two segments [34]

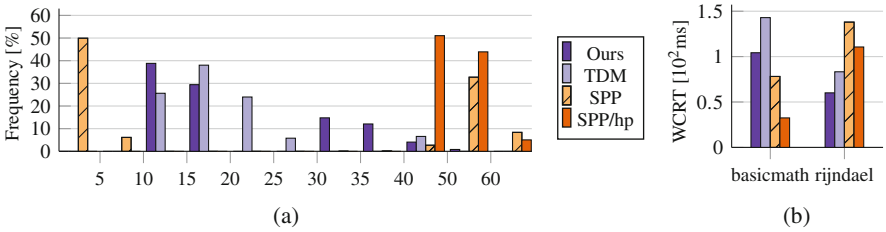


Fig. 10 *Basicmath* and *rijndael* as replicated tasks in DMR running on a dual-core configuration with 20.2% load (5% load from independent tasks) [34]. (**a**) WCRT of independent tasks [ms]. (**b**) WCRT of FJ tasks

to medium loads. SPP provides very short response times with lower loads but, as the load increases, the schedulability drops fast due to high interference (and thus high WCRT) suffered by fork-join tasks. For reference purposes, we also plot the schedulability of SPP when assigning the highest priorities to the fork-join tasks (SPP/hp). The schedulability in higher loads improves but losing deadline monotonicity guarantees renders the systems unusable in practice. Moreover, when increasing the jitter to 20% (relative to period), schedulability decreases 14.2% but shows the same trends for all schedulers.

Figure 10 details the tasks' WCRTs when the system load is 20.2%. Indeed, when schedulable, SPP provides some of the shortest WCRTs for independent tasks, and SPP/hp improves the response times of fork-join tasks at the expense of the independent tasks'. Our approach provides a balanced trade-off between the performance of independent tasks and of fork-join tasks, and achieves high schedulability even in higher loads.

5.2 Evaluation with Synthetic Workload

We now evaluate the performance of our approach when varying parameters such as stage length and cycle Φ.

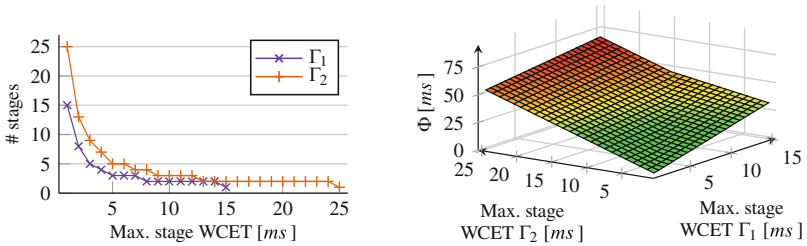


Fig. 11 Parameters of two fork-join tasks Γ_1 and Γ_2 with two segments running on a dual-core configuration [34]. (**a**) Stages of Γ_1 and Γ_2. (**b**) Cycle Φ

5.2.1 Evaluation of Fork-Join Tasks

Two fork-join tasks Γ_1 and Γ_2 with two segments each (i.e., replicas in DMR) are in DMR) are mapped to two cores. The total WCETs[3] of Γ_1 and Γ_2 are 15 and 25ms, respectively. Both tasks are sporadic, with a minimum distance of 1s between activations. The number of stages of Γ_1 and Γ_2 is varied as a function of the maximum stage WCET, as depicted in Fig. 11a. The length of the cycle Φ, depicted in Fig. 11b, varies with the maximum stage WCET since it is derived from them (cf. Sect. 3.4).

The system performance as the maximum stage lengths of Γ_1 and Γ_2 increase is reported in Fig. 12. The WCRT of Γ_1 increases with the stage length (Fig. 12a) as it depends on the number of stages and Φ's length. In fact, the WCRT of Γ_1 is longest when the stages of Γ_1 are the shortest and the stages of the interfering fork-join task (Γ_2) are the longest. Conversely, WCRT of Γ_1 is shortest when its stages are the longest and the stages of the interfering fork-join task are the shortest. The same occurs to Γ_2 in Fig. 12b. Thus, there is a trade-off between the response times of interfering fork-join tasks. This is plotted in Fig. 13 as the sum of the WCRTs of Γ_1 and Γ_2. As can be seen in Fig. 13, low response times can be obtained next and above to the line segment between the origin (0, 0, 0) and the point (15, 25, 0), the total WCETs[1] of Γ_1 and Γ_2, respectively.

5.2.2 Evaluation of Independent Tasks

To evaluate the impact of the parameters on independent tasks, we extend the previous scenario introducing 25% load on each core with ten independent tasks generated with UUniFast [7]. The task periods are within the interval [15, 500] ms for the first experiment, and the interval [25, 500] ms for the second. The priority

[3]The sum of the WCET of all stages of a fork-join task.

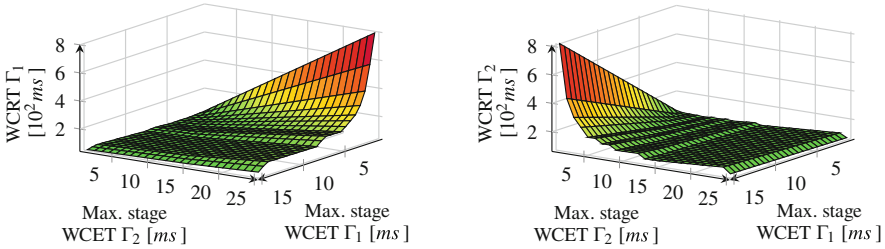


Fig. 12 Performance of fork-join tasks Γ_1 and Γ_2 as a function of the maximum stage WCET [34]. (**a**) WCRT of Γ_1. (**b**) WCRT of Γ_2

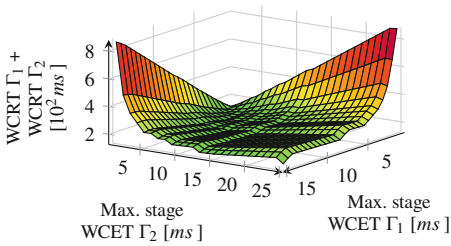


Fig. 13 WCRT trade-off between interfering fork-join tasks [34]

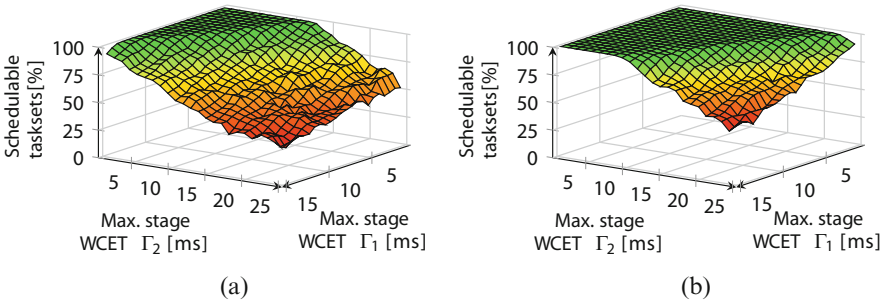


Fig. 14 Schedulable tasksets as a function of the maximum stage WCET of fork-join tasks Γ_1 and Γ_2 with 25% load from independent tasks [34]. (**a**) Task period interval [15–500] ms. (**b**) Task period interval [25–500] ms

assignment is deadline monotonic and considers that the deadline is equal to the period.

The schedulability as a function of the stage lengths is shown in Fig. 14. Sufficiently long stages cause the schedulability to decrease as independent tasks with short periods start missing their deadlines. This is seen in Fig. 14a when the stage length of either fork-join task reaches 15 ms, the minimum period for the generated tasksets. Thus, when increasing the minimum period of generated tasks to 25 ms, the number of schedulable tasksets also increases (Fig. 14b).

The maximum stage length of a fork-join task has direct impact on the response times and schedulability of the system. For the sake of performance, shorter stage

lengths are preferred. However, that is not always possible because it would result in a large number of stages or because of the application, which restricts the minimum stage length (cf. Sect. 5.1.1). Nonetheless, fork-join tasks still are able to perform well with appropriate parameter choices. Additionally, one can formulate the problem of finding the stage lengths according to an objective function, such as minimize the overall response time or maximize the slack. The offsets can also be included in the formulation, as long as Constraints 1 and 2 are met.

6 Conclusion

This chapter started with an overview of the project ASTEROID. ASTEROID developed a cross-layer fault-tolerance approach to provide reliable software execution on unreliable hardware. The approach is based on replicated software execution and exploits the large number of cores available in modern and future architectures at a higher level of abstraction without resorting to the inefficient hardware redundancy. The chapter then focused on the performance of replicated execution and the replica-aware co-scheduling, which was developed in ASTEROID.

The replica-aware co-scheduling for mixed-critical systems, where applications with different requirements and criticalities co-exist, overcomes the performance limitations of standard schedulers such as SPP and TDM. A formal WCRT analysis was presented, which supports different recovery strategies and accounting for the NoC communication delay and overheads due to replica management and state comparison. The replica-aware co-scheduling provides for high worst-case performance of replicated software execution on many-core architectures without impairing the remaining tasks in the system. Experimental results with benchmark applications showed an improvement on taskset schedulability of up to 6.9× when compared to Partitioned SPP and 1.5× when compared to a TDM-based scheduler.

References

1. Andersson, B., de Niz, D.: Analyzing Global-EDF for multiprocessor scheduling of parallel tasks. In: International Conference On Principles Of Distributed Systems, pp. 16–30. Springer, Berlin (2012)
2. Axer, P.: Performance of time-critical embedded systems under the influence of errors and error handling protocols. Ph.D. Thesis, TU Braunschweig (2015)
3. Axer, P., Sebastian, M., Ernst, R.: Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2011)
4. Axer, P., Sebastian, M., Ernst, R.: Probabilistic response time bound for CAN messages with arbitrary deadlines. In: 2012 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1114–1117. IEEE, Piscataway (2012)

5. Axer, P., Ernst, R., Döbel, B., Härtig, H.: Designing an analyzable and resilient embedded operating system. In: Proceedings of Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES'12), Braunschweig (2012)
6. Axer, P., Quinton, S., Neukirchner, M., Ernst, R., Dobel, B., Hartig, H.: Response-time analysis of parallel fork-join workloads with real-time constraints. In: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'13) (2013)
7. Bini, E., Buttazzo, G.C.: Measuring the performance of schedulability tests. Real-Time Syst. **30**(1–2), 129–154 (2005)
8. Binkert, N., Beckmann, B., Black, G., et al.: The Gem5 simulator. SIGARCH Comput. Archit. News **39**(2) (2011). https://doi.org/10.1145/2024716.2024718
9. Döbel, B., Härtig, H.: Who watches the watchmen? – protecting operating system reliability mechanisms. In: International Workshop on Hot Topics in System Dependability (HotDep'12) (2012)
10. Döbel, B., Härtig, H.: Where have all the cycles gone? Investigating runtime overheads of OS-assisted replication. In: Proceedings of Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES'13), pp. 2534–2547 (2013)
11. Döbel, B., Härtig, H.: Can we put concurrency back into redundant multithreading? In: Proceedings of the International Conference on Embedded Software (EMSOFT'14) (2014)
12. Döbel, B., Härtig, H., Engel, M.: Operating system support for redundant multithreading. In: Proceedings of the International Conference on Embedded Software (EMSOFT'12) (2012)
13. Engel, M., Döbel, B.: The reliable computing base-a paradigm for software-based reliability. In: Proceedings of Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES'12), pp. 480–493 (2012)
14. Feitelson, D.G., Rudolph, L.: Gang scheduling performance benefits for fine-grain synchronization. J. Parallel Distrib. Comput. **16**(4), 306–318 (1992)
15. Gaillard, R.: Single Event Effects: Mechanisms and Classification. In: Soft Errors in Modern Electronic Systems. Springer, New York (2011)
16. Goossens, J., Berten, V.: Gang ftp scheduling of periodic and parallel rigid real-time tasks (2010). arXiv:1006.2617
17. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., et al.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(1), 8–23 (2012)
18. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4. 2001) (2001)
19. Härtig, H., Roitzsch, M., Weinhold, C., Lackorzynski, A.: Lateral thinking for trustworthy apps. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1890–1899. IEEE, Piscataway (2017)
20. Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System Level Performance Analysis–the SymTA/S Approach. IEE Proc.-Comput. Digit. Tech. **152**, 148–166 (2005)
21. 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., Teich, J., Wehn, N., Wunderlich, H.J.: Design and architectures for dependable embedded systems. In: Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '11, pp. 69–78. ACM, New York (2011). http://doi.acm.org/10.1145/2039370.2039384
22. 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)
23. Hoffmann, M., Lukas, F., Dietrich, C., Lohmann, D.: dOSEK: the design and implementation of a dependability-oriented static embedded kernel. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15) (2015)
24. International Standards Organization: ISO 26262: Road Vehicles – Functional Safety (2011)

25. Kaiser, R., Wagner, S.: Evolution of the PikeOS microkernel. In: First International Workshop on Microkernels for Embedded Systems (2007)
26. Kato, S., Ishikawa, Y.: Gang EDF scheduling of parallel task systems. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09) (2009)
27. Lehoczky, J.: Fixed priority scheduling of periodic task sets with arbitrary deadlines. In: Proceedings 11th Real-Time Systems Symposium (RTSS'90) (1990)
28. Leuschner, L., Küttler, M., Stumpf, T., Baier, C., Härtig, H., Klüppelholz, S.: Towards automated configuration of systems with non-functional constraints. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotDep'17), pp. 111–117. ACM, New York (2017)
29. NXP MPC577xK Ultra-Reliable MCU Family (2017). http://www.nxp.com/assets/documents/data/en/fact-sheets/MPC577xKFS.pdf
30. Ousterhout, J.K.: Scheduling techniques for concurrent systems. In: International Conference on Distributed Computing (ICDCS), vol. 82, pp. 22–30 (1982)
31. Palencia, J.C., Harbour, M.G.: Schedulability analysis for tasks with static and dynamic offsets. In: Proceedings 19th IEEE Real-Time Systems Symposium (RTSS'98) (1998)
32. Rambo, E.A., Ernst, R.: Providing flexible and reliable on-chip network communication with real-time constraints. In: 1st International Workshop on Resiliency in Embedded Electronic Systems (REES) (2015)
33. Rambo, E.A., Ernst, R.: Worst-case communication time analysis of networks-on-chip with shared virtual channels. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'15) (2015)
34. Rambo, E.A., Ernst, R.: Replica-aware co-scheduling for mixed-criticality. In: 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), vol. 76, pp. 20:1–20:20 (2017). https://doi.org/10.4230/LIPIcs.ECRTS.2017.20
35. Rambo, E.A., Ahrendts, L., Diemer, J.: FMEA of the IDAMC NoC. Tech. Rep., Institute of Computer and Network Engineering – TU Braunschweig (2013)
36. Rambo, E.A., Tschiene, A., Diemer, J., Ahrendts, L., Ernst, R.: Failure analysis of a network-on-chip for real-time mixed-critical systems. In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE) (2014)
37. Rambo, E.A., Tschiene, A., Diemer, J., Ahrendts, L., Ernst, R.: FMEA-based analysis of a network-on-chip for mixed-critical systems. In: Proceedings of the Eighth IEEE/ACM International Symposium on Networks-on-Chip (NOCS'14) (2014)
38. Rambo, E.A., Saidi, S., Ernst, R.: Providing formal latency guarantees for ARQ-based protocols in networks-on-chip. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'16) (2016)
39. Rambo, E.A., Seitz, C., Saidi, S., Ernst, R.: Designing networks-on-chip for high assurance real-time systems. In: Proceedings of the IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC'17) (2017)
40. Rambo, E.A., Seitz, C., Saidi, S., Ernst, R.: Bridging the gap between resilient networks-on-chip and real-time systems. IEEE Trans. Emer. Top. Comput. **8**, 418–430 (2020). http://doi.org/10.1109/TETC.2017.2736783
41. Richter, K.: Compositional scheduling analysis using standard event models. Ph.D. Thesis, TU Braunschweig (2005)
42. RTCA Incorporated: DO-254: Design Assurance Guidance For Airborne Electronic Hardware (2000)
43. Tindell, K., Burns, A., Wellings, A.: An extendible approach for analyzing fixed priority hard real-time tasks. Real-Time Syst. **6**(2), 133–151 (1994)

Dependability Aspects in Configurable Embedded Operating Systems

Horst Schirmeier, Christoph Borchert, Martin Hoffmann, Christian Dietrich, Arthur Martens, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk

1 Introduction

Future hardware designs for embedded systems will exhibit more parallelism and energy efficiency at the price of being less reliable, due to shrinking structure sizes, increased clock frequencies, and lowered operating voltages [9]. In embedded control systems, the handling of soft errors—e.g., transient bit flips in the memory hierarchy—is becoming mandatory for all safety integrity level (SIL) 3 or SIL 4 categorized safety functions [30, 35]. Established solutions stem mostly from the avionics domain and employ extensive hardware redundancy or specifically hardened hardware components [55]—both of which are too costly to be deployed in commodity products.

H. Schirmeier (✉)
Embedded System Software Group, TU Dortmund, Dortmund, Germany
e-mail: horst.schirmeier@tu-dortmund.de

C. Borchert · O. Spinczyk
Embedded Software Systems Group, Osnabrück University, Osnabrück, Germany
e-mail: christoph.borchert@uos.de; olaf.spinczyk@uos.de

M. Hoffmann
System Software Group, FAU Erlangen, Erlangen, Germany
e-mail: hoffmann@cs.fau.de

C. Dietrich · D. Lohmann
Systems Research and Architecture Group, Leibniz University Hannover, Hannover, Germany
e-mail: dietrich@sra.uni-hannover.de; lohmann@sra.uni-hannover.de

A. Martens · R. Kapitza
Institute of Operating Systems and Computer Networks, TU Braunschweig, Braunschweig, Germany
e-mail: martens@ibr.cs.tu-bs.de; kapitza@ibr.cs.tu-bs.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_4

Software-based redundancy techniques, especially redundant execution with majority voting in terms of TMR, are well-established countermeasures against soft errors on the application level [24]. By combining them with further techniques—such as arithmetic codes—even the voter as the single point of failure (SPOF) can be eliminated [53]. However, all these techniques "work" only under the assumption that the application is running on top of a soft-error-resilient system-software stack.

In this chapter, we address the problem of software-stack hardening for three different points in the system-software and fault-tolerance technique design space:

- In Sect. 3 we investigate soft-error hardening techniques for a statically configured OS, which implements the automotive OSEK/AUTOSAR real-time operating system (RTOS) standard [5, 40]. We answer the research question what the *general reliability limits* in this scenario are when aiming at *reliability as a first-class design goal*. We show that harnessing the static application knowledge available in an AUTOSAR environment, and protecting the OS kernel with AN-encoding, yields an extremely reliable software system.
- In Sect. 4 we analyze how programming-language and compiler extensions can help to modularize fault-tolerance mechanisms. By applying the resulting fault-tolerance modules to a *dynamic* commercial off-the-shelf (COTS) embedded OS, we explore how far reliability can be pushed when a legacy software stack needs to be maintained. We show that aspect-oriented programming (AOP) is suitable for encapsulating generic software-implemented hardware fault tolerance (SIHFT) mechanisms, and can improve reliability of the targeted software stack by up to 79%.
- Looking beyond bit flips in the memory hierarchy, in Sect. 5 we investigate how a system-software stack can survive even more adverse fault models such as whole-system outages. Using persistent memory (PM) technology for state conservation, our findings include that software transactional memory (STM) facilitates maintaining state consistency and allows fast recovery.

These works have been previously published in conference proceedings and journals [8, 29, 36], and are presented here in a summarized manner. Section 6 concludes the chapter and summarizes the results of the *DanceOS* project, which was funded by the German Research Foundation (DFG) over a period of 6 years as part of the priority program SPP 1500 "Dependable Embedded Systems" [26] (Fig. 1).

2 Related Work

Dependable Embedded Operating Systems While most work from the dependable-systems community still assumes the OS itself to be too hard to protect, the topic of RTOS reliability in case of transient faults has recently gained attention. The C^3 µ-kernel tracks system-state transitions at the inter-process communication (IPC) level to be able to recover system components in case of a fault [50]. Their approach,

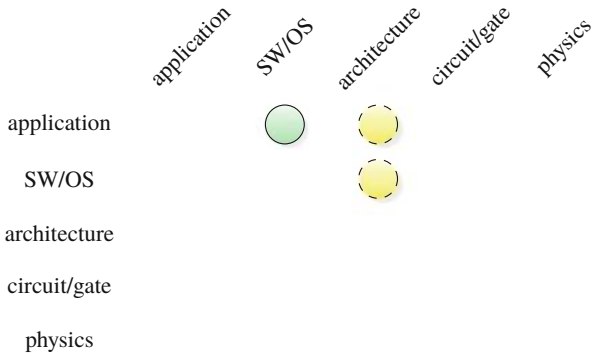


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

however, assumes that faults are detected immediately and never turn into silent data corruptions (SDCs), and that the recovery functionality itself is part of the RCB. L4/Romain [19] employs system-call interception to provide transparent thread-level TMR—and, hence, error detection,—but still requires a reliable µ-kernel. The hypervisor approach of Quest-V [34] reduces the *software*-part of the RCB even further—at the price of increasing the *hardware*-part for the required virtualization support. In the end, however, all these approaches assume early and reliable detection of faults and their strict containment inside the RCB, which our three approaches provide.

Software-Based Soft-Error Detection and Correction The concept of AN-encoding has been known for quite a while and has been taken up in recent years in compiler- and interpreter-based solutions [45]. Yet, these generic realizations are not practicable for realizing a RCB—not only due their immense runtime overhead of a factor of 10^3 up to 10^5, but also due to the specific nature of low-level system software. Thus, following our proven CoRed concept [28], we concentrate the encoded execution to the *minimal necessary points*. Besides AN-encoding, several more generic *error detection and recovery mechanisms* (EDMs/ERMs) exist and have been successfully deployed. Shirvani et al. [48] evaluate several software-implemented error-correcting codes for application in a space satellite to obviate the use of a low-performance radiation-hardened CPU and memory. Read-only data segments are periodically scrubbed to correct memory errors, whereas protected *variables* must be accessed manually via a special API to perform error correction. Similarly, *Samurai* [41] implements a C/C++ dynamic memory allocator with a dedicated API for access to replicated heap memory. Programmers have to manually invoke functions to check and update the replicated memory chunks. The latter approach exposes the heap allocator as single point of failure, which is not resilient against memory errors. To automate the hardening process, some works extend *compilers* for transforming code to add fault tolerance [44]. These

approaches are based on duplicating or even triplicating important *variables* of single-threaded user-level programs. Our work differs in that we use the general-purpose AspectC++ compiler that allows us to focus on the implementation of software-based EDM/ERMs in the OS/application layer, instead of implementing special-purpose compilers. AOP also allows to separate the "business logic" from fault-tolerance implementations, which has, e.g., been pioneered by Alexandersson et al. [2]—however at the cost of 300% runtime overhead.

State Consistency in Non-volatile Memories Maintaining state consistency in persistent memory has been achieved on the level of process-wide persistence [10, 39] and specialized file systems [13, 20]. Our *DNV Memory* approach shares the most similarities with libraries that provide safe access to a persistent heap [6, 12, 54]. Mnemosyne [54] shows the overall steps that are needed to build a persistent heap, while NV-Heaps [12] focuses mainly on usability aspects. Both libraries rely on a transactional-memory model that stores logs in persistent memory and executes expensive flush operations to ensure data consistency in presence of power failures. In order to improve performance, the memory allocator of Makalu [6] guarantees the consistency of its own meta data without the need of transactions. However, it does not extend this ability to the data stored within. Thus, library support, similar to Mnemosyne [54], is still needed to enforce durability. *DNV Memory* shares with these approaches the transactional model and the goal to provide a persistent heap, but aims at improving performance and lifetime of persistent applications by reducing the amount of writes to persistent memory. Additionally, *DNV Memory* provides transparent dependability guarantees that none of the previous work has covered.

3 *d*OSEK: A Dependable RTOS for Automotive Applications

In the following, we present the design and implementation of *d*OSEK, an OSEK/AUTOSAR-conforming [5, 40] RTOS that serves as reliable computing base (RCB) for safety-critical systems. *d*OSEK has been developed from scratch with dependability as the first-class design goal based on a two-pillar design approach: First we aim for strict *fault avoidance*[1] by an in-depth static tailoring of the kernel towards the concrete application and hardware platform—without restricting the required RTOS services. Thereby, we constructively minimize the (often redundant) vulnerable runtime state. The second pillar is to then constructively reintegrate redundancy in form of dependability measures to eliminate the remaining SDCs in the essential state. Here, we concentrate—in contrast to others [4, 50]—on reliable *fault detection* and fault containment within the kernel execution path (Sect. 3.2) by

[1] Strictly speaking, we aim to avoid *errors* resulting from transient hardware faults.

employing arithmetic encoding [23] to realize self-contained data and control-flow error detection across the complete RTOS execution path.

We evaluate our hardened *d*OSEK against ERIKA [21], an industry-grade open-source OSEK implementation, which received an official OSEK/VDX certification (Sect. 3.3). We present the runtime and memory overhead as well as the results of extensive fault-injection campaigns covering the *complete* fault space of single-bit faults in registers and volatile memory. Here, *d*OSEK shows an improvement of four orders of magnitude regarding the SDC count, compared to ERIKA.

3.1 Development of a Fault-Avoiding Operating System

Essentially, a transient fault can lead to an error inside the kernel only if it affects either the kernel's control or data flow. For this, it has to hit a memory cell or register that carries *currently alive* kernel state, such as a global variable (always alive), a return address on the stack (alive during the execution of a system call), or a bit in the status register of the CPU (alive only immediately before a conditional instruction). Intuitively, the more long-living state a kernel maintains, the more prone it is to transient faults. Thus, our first rule of fault-avoiding OS development is: ❶ **Minimize the time spent in system calls and the amount of volatile state, especially of global state that is alive *across* system calls.**

However, no kernel can provide useful services without any runtime state. So, the second point to consider is the containment and, thus, detectability of data and control-flow errors by local sanity checks. Intuitively, bit flips in pointer variables have a much higher error range than those used in arithmetic operations; hence, they are more likely to lead to SDCs. In a nutshell, any kind of indirection at runtime (through data or function pointers, index registers, return addresses, and so on) impairs the inherent robustness of the resulting system. Thus, our second rule of fault-avoiding operating-system development is: ❷ **Avoid indirections in the code and data flow.**

In *d*OSEK, we implement these rules by an extensive static analysis of the application code followed by a subsequent dependability-oriented "pointer-less" generation of the RTOS functionality. Our approach follows the OSEK/AUTOSAR system model of static tailoring [5, 40], which in itself already leads to a significant reduction of state and SDC vulnerability [27]. We amplify these already good results by a flow-sensitive analysis of all application–RTOS interactions [17, 18] in order to perform a partial specialization of system calls: Our system generator specializes each system call *per invocation* to embed it into the particular application code. This facilitates an aggressive folding of parameter values into the code. Therefore, less state needs to be passed in volatile registers or on the stack (rule ❶). We further achieve a pointer-less design by allocating all system objects statically as global data structures, with the help of the generator. In occasions where pointers would be used to select one object out of multiple possible candidates, an array at a constant address with small indices is preferred (rule ❷).

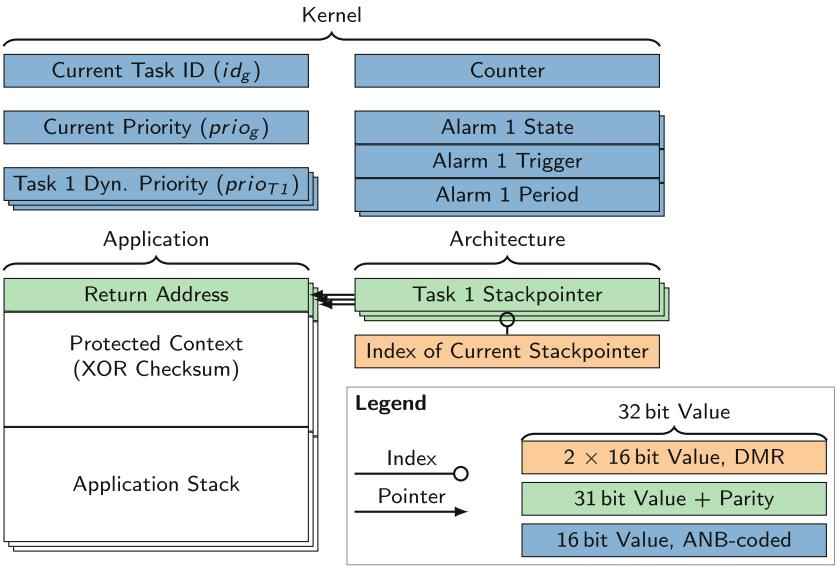


Fig. 2 Overview of the OS data kept in RAM of an example system composed of three tasks and two alarms. Each box represents a 32-bit memory location. All kernel data are hardened using an ANB-Code. The remaining application- and architecture-specific values are safeguarded by dual modular redundancy (DMR) or parity bits

Figure 2 depicts the resulting state of this analysis by the example of a system consisting of three tasks and two alarms: The remaining volatile state variables are subsumed under the blocks *Application*, *Architecture*, and *Kernel*. The architecture-independent minimal *Kernel* state is condensed to two machine words for the current task's priority, its id, and one machine word per task for the task's dynamic priority according to the priority ceiling protocol. Depending on the requirements of the application, the kernel maintains the current state of additional resources: in this case two alarms (three machine words each) and one counter (one machine word). The *Architecture* blocks are related to the dispatching mechanism of the underlying processor. In case of the IA-32, this is reduced to the administration of one stack pointer per task.

The most frequently used (but far less visible) pointers are the stack pointer and the base pointer. Albeit less obvious, they are significant: A corrupted stack pointer influences all local variables, function arguments, and the return address. Here, we eliminated the indirection for local variables by storing them as static variables at fixed, absolute addresses, while keeping isolation in terms of visibility and memory protection (rule ❷). Furthermore, by aggressively inlining the specialized system calls into the application code, we reduce the spilling of parameter values and return addresses onto the vulnerable stack, while keeping the hardware-based spatial isolation (MPU/MMU-based AUTOSAR memory protection) between applications and kernel using inline traps [15] (rule ❶).

3.2 Implementing a Fault-Detecting Operating System

*d*OSEK's *fault-detection* strategies can be split up into two complementary concepts: First, coarse-grained hardware-based fault-detection mechanisms, mainly by means of MPU-based memory and privilege isolation. Second, fine-grained software-based concepts that protect the kernel-internal data/control flows.

Hardware-based isolation by watchdogs and memory protection units (MPUs) are a widely used and a proven dependability measure. Consequently, *d*OSEK integrates the underlying architecture's mechanisms into its system design, leveraging a coarse-grained fault detection between tasks and the kernel. We furthermore employ hardware-based isolation to minimize the set of kernel-writable regions during a system call, which leverages additional error-detection capabilities for faulty memory writes from the kernel space. With our completely generative approach, all necessary MPU configurations can be derived already at compile time and placed in robust read-only memory (ROM).

The execution of the *d*OSEK kernel itself is hardened with a fine-grained arithmetic encoding. All kernel data structures are safeguarded using a variant of an AN-code [23] capable of detecting both data- and control-flow errors. The code provides a constant common key A, allowing to uncover errors when calculating the remainder, and a variable-specific, compile-time constant signature B_n detecting the mix-up of two encoded values as well as the detection of faulty control flows—the ANB-Code:

$$\underset{\text{Encoded Value}}{n_{enc}} = \underset{\text{Key}}{A} \cdot \underset{\text{Value}}{n} + \underset{\text{Signature}}{B_n}$$

A particular feature of arithmetic codes is a set of code-preserving arithmetic operations, which allow for computation with the encoded values. Hence, a continuous sphere of redundancy is spanned, as the corresponding operands remain encoded throughout the entire kernel execution.

In addition to the existing elementary arithmetic operations, *d*OSEK also requires an encoded variant of the mandatory OSEK/AUTOSAR fixed-priority scheduling algorithm [40]: The encoded scheduler is based on a simple prioritized task list. Each task's current dynamic priority is stored at a fixed location (see also Fig. 2), with the lowest possible value, an encoded zero, representing the suspended state. To determine the highest-priority task, the maximum task priority is searched by comparing all task priorities sequentially. Thus, the algorithm's complexity in space and time is linear to the constant number of tasks. Figure 3 shows the basic concept for three tasks: The sequence processes a global tuple of ANB-encoded values storing the current highest-priority task id found so far, and the corresponding priority ($\langle \text{id}_g, \text{prio}_g \rangle$, see Fig. 2). Sequential compare-and-update operations, based on an encoded greater-equal decision on a tuple of values (*ge_tuple*), compare

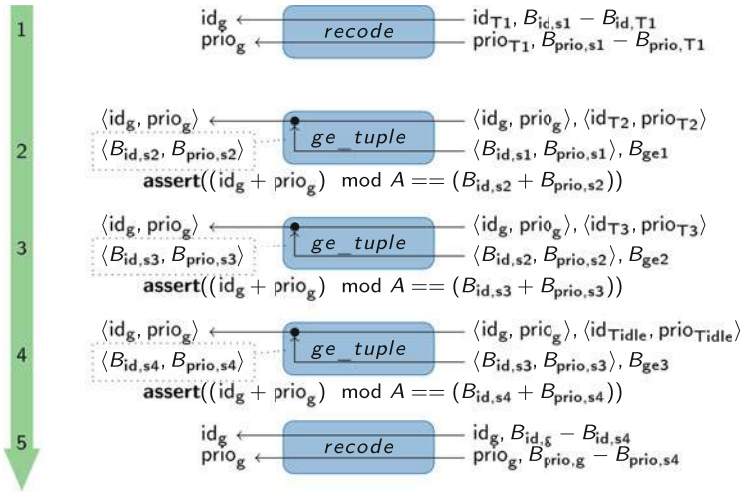


Fig. 3 General sequence of the encoded scheduling operation on the example of three tasks (T1, T2, T3). All operations on signatures B are calculated already at compile time

the tuples' priority value and update the global values, if necessary. The sequence consists of five steps, as shown in Fig. 3:

(1) Initialize prio_g and id_g to the first task.
(2–3) For all further tasks, compare the task's priority to prio_g: If greater or equal, update $\langle \text{id}_g, \text{prio}_g \rangle$.
(4) Repeat the last step for the idle task.
(5) Recode the results to their original signatures.

The idle task priority is constantly bound to an encoded zero that is representing a suspended state. Thus, if all previous tasks are suspended, the last comparison (in step 4) will choose the idle task halting the system until the next interrupt.

Aside from the actual compare-and-update operation on fully encoded values, the *ge_tuple* function additionally integrates control-flow error detection. For each step, all signatures of the input operands ($B_{id,s1..s4}$, $B_{prio,s1..s4}$) and the signature of the operation itself ($B_{ge1..4}$) are merged into the resulting encoded values of the global tuple. Each corresponding signature of a step is then applied in the next operation accordingly. Thus, the dynamic values of the result tuple accumulate the signatures of all preceding operations. As the combination of these compile-time constant signatures is known before runtime, interspersed assertions can validate the correctness of each step. Even after the final signature recode operation (step 5), any control-flow error is still detectable by the dynamic signature. Thus, the correctness of the encoded global tuple can be validated at any point in time. In effect, fault detection is ensured, as all operations are performed on encoded values.

The remaining dynamic state highly depends on the underlying architecture. Regarding the currently implemented IA-32 variant, we were able to reduce this

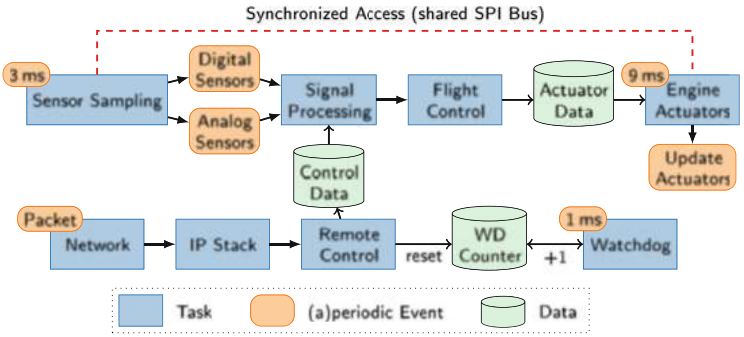


Fig. 4 Simplified representation of the *I4Copter* task and resource constellation used as evaluation scenario

runtime state to an array storing the stack pointers of preempted tasks, and an corresponding index variable, as shown in Fig. 2. The variables are used within each interrupt entry as well as during the actual dispatch operation. As they are not involved in any arithmetic calculations, but only read and written, we can avoid the overhead of the ANB-encoding in these cases and protect them by DMR or parity checks, respectively.

3.3 Evaluation

For comparison, we chose ERIKA Enterprise [21], an industry-grade (i.e., formally certified) open-source implementation of the automotive OSEK standard [40].

The evaluation is based on a realistic system workload scenario considering all essential RTOS services, resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application (cf. Fig. 4). The scenario consists of 11 tasks, which are activated either periodically or sporadically by one of four interrupts. Inter-task synchronization is done with OSEK resources and a watchdog task, observing the remote control communication. We evaluated several variants of ERIKA and *d*OSEK, all running the same task set. As ERIKA does not provide support for hardware-based memory protection, we also disabled the MPU in *d*OSEK:

ERIKA Standard version of ERIKA with enabled sanity checks (SVN r3274).

*d*OSEK (unprotected) For the *d*OSEK base version only the indirection avoidance and the generative approach are used against SDCs.

*d*OSEK (FT) The safeguarded kernel execution with encoded operations.

*d*OSEK (FT+ASS) Like FT, but with additional assertions obtained by a flow-sensitive global control-flow analysis [18].

The application flow is augmented with 172 checkpoints. Every RTOS under test executes the application for three hyper periods, while, at the same time a trace of visited checkpoints is recorded. It is the mission of the systems under test to reproduce this sequence, without corrupting the application state. If the sequence *silently* diverges in the presence of faults, we record a silent data corruption.[2] The application state (task stacks) is checked for integrity at each checkpoint. To evaluate the fault containment within the kernel execution, we further recorded an SDC in case of violated integrity. Both SDC detection mechanisms were realized externally by the FAIL* fault-injection framework [47] without influencing the runtime behavior of the systems under test. Since FAIL* has the most mature support for IA-32, we choose this architecture as our evaluation platform. FAIL* provides elaborate fault-space pruning techniques that allow to cover the *entire* space of effective faults, while keeping the total number of experiments manageable. The evaluated fault space includes all *single-bit faults* in the *main memory*, in the *general-purpose registers*, the *stack pointer, and flags registers*, as well as the *instruction pointer*.

3.3.1 Fault-Injection Results

All OS variants differ in code size, runtime, and memory consumption—parameters that directly influence the number of effective injected faults. To directly compare the robustness independent of any other non-functional properties, we concentrate on the resulting absolute SDC count, which represents the number of cases in which the RTOS did not provide the expected behavior. Figure 5 shows, on a logarithmic scale, the resulting SDC counts.

The results show that, compared to ERIKA, the *unprotected d*OSEK variant already faces significantly fewer control-flow and register errors. This is caused by the means of constructive fault avoidance, particularly the avoidance of indirections in the generated code. The activation of fault tolerance measures (*d*OSEK *FT*) significantly reduces the number of memory errors, which in total reduces the SDC count compared to ERIKA by **four orders of magnitude**. The remaining SDCs can further be halved by adding static assertions (*d*OSEK *FT+ASS*).

3.3.2 Memory- and Runtime Costs

On the downside, aggressive inlining to avoid indirections, but especially the encoded scheduler and kernel execution path leads to additional runtime and memory costs, which are summarized in Table 1. Compared again to ERIKA, the SDC reduction by four orders of magnitude is paid for with a 4× increase in runtime and a 20× increase in code size. As most of the code bloat is caused by the inlining

[2] Faults that lead to a hardware trap are *not* counted as silent, as they are handled by the kernel.

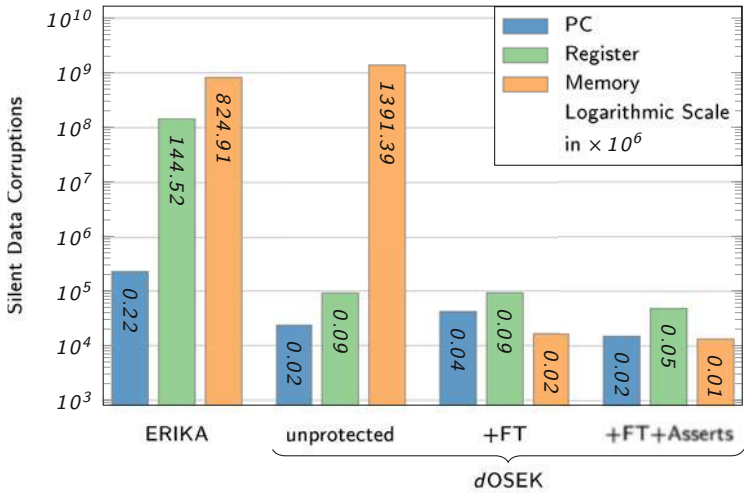


Fig. 5 SDC distribution for the evaluated variants of the *I4Copter* scenario (Fig. 4 on a logarithmic scale; pruned experiments are factored in). The encoded *d*OSEK system achieves an improvement in the SDC count by four orders of magnitude compared to ERIKA (base)

Table 1 Memory- and runtime cost

System	Code size (bytes)	Runtime (instructions)
ERIKA	3782	38,912
*d*OSEK (unprotected)	14,985	29,223
*d*OSEK FT	53,956	110,524
*d*OSEK FT+ASS	71,049	121,583
*d*OSEK FT+ASS+OPT	24,955	90,106

of the encoded scheduler at each call site, we have added a fifth variant (*d*OSEK *FT+ASS+OPT*) that employs further whole-program static optimizations to exclude unnecessary scheduler invocations (see [17] for further details). This version is still $10^4\times$ less vulnerable to SDCs, but reduces the runtime overhead to $2.5\times$ and the code overhead to $8\times$.

4 Modularizing Software-Based Memory Error Detection and Correction

The *d*OSEK approach in the previous section showed the general reliability limits when designing a static OS from scratch, focusing on reliability as a first-class design goal. However, a different and quite common use case is that the requirements entail using a preexisting COTS embedded OS, which is often *dynamic* in the sense that it provides an interface for creating and destroying threads or memory

allocations at runtime. To protect this class of system-software stacks against transient hardware faults—e.g., bit flips—in memory, we propose a software-based memory-error recovery approach that exploits application knowledge about memory accesses, which are analyzed at compile time and hardened by compiler-generated runtime checks.

A central challenge is the *placement* of these runtime checks in the control flow of the software, necessitating an analysis that determines which program instructions access which parts of the memory. In general, this is an undecidable problem for pointer-based programming languages; however, if we assume an *object-oriented programming model*, we can reason that non-public data-structure members are accessed only within member functions of the same class. Consequently, data structures—or, *objects*—can be examined for errors by inserting a runtime check *before* each member-function call.

In this section, we describe our experiences with devising such an object-level error recovery in AspectC++ [51]—an AOP extension to C++,—and applying it to the embedded Configurable operating system (eCos) [37]. Our software-based approach, called Generic Object Protection (GOP), offers the flexibility to choose from an extensible toolbox of error-detecting and error-correcting codes, for example, CRC and Hamming codes.

4.1 Generic Object Protection with AspectC++

Our experience with the embedded operating system eCos shows that OS kernel data structures are highly susceptible to soft errors in main memory [8]. Several kernel data structures, such as the process scheduler, persist during the whole OS uptime, which increases the chance of being hit by a random soft error.

As a countermeasure, OS kernel data structures can contain redundancy, for example, a separated Hamming code [48]. Before an instance of such a data structure—an *object* in object-oriented jargon—is used, the object can be examined for errors. Then, after object usage, the Hamming code can be updated to reflect modifications of the object.

Manually implementing such a protection scheme in an object-oriented programming language is a tedious and error-prone task, because *every* program statement that operates on such an object needs careful manipulation. Therefore, we propose to integrate object checking into existing source code by AOP [32]. Over the last 19 years, we have developed the general-purpose *AspectC++* programming language and compiler [51] that extends C++ by AOP features. A result of the SPP-1500's *DanceOS* project is AspectC++ 2.0, which provides new language features that allow for a completely modular implementation of the sketched object protection scheme—the GOP. In the following, we describe these programming-language features taking the example of GOP.

```
1  aspect GenericObjectProtection {
2    pointcut critical() = "Cyg_Scheduler" || "Cyg_Thread"; // list of types
3    advice critical() : slice class { // generic class extension
4      HammingCode<JoinPoint> code; // template meta-program
5    };
6
7    advice construction(critical()) : after() {
8      tjp->target()->code.update(); // generic advice
9    }
10
11   pointcut trigger_check() = call(member(critical())) ||
12     get(member(critical())) || set(member(critical()));
13   pointcut trigger_update() = call(member(critical())) ||
14     set(member(critical()));
15
16   advice trigger_check() : before() {
17     if (tjp->that() != static_cast<void*>(tjp->target())) {
18       tjp->target()->code.check(); // check callee
19   }}
20   advice trigger_update() : after() {
21     if (tjp->that() != static_cast<void*>(tjp->target())) {
22       tjp->target()->code.update(); // update callee
23   }}
24
25   pointcut any_call() = call("% ...::%(...)");
26   advice any_call() && within(member(critical())) : before() {
27     if (tjp->that() != static_cast<void*>(tjp->target())) {
28       tjp->that()->code.update(); // update caller
29   }}
30   advice any_call() && within(member(critical())) : after() {
31     if (tjp->that() != static_cast<void*>(tjp->target())) {
32       tjp->that()->code.check(); // check caller
33   }}
34 };
```

Fig. 6 A simplified implementation of the GOP mechanism written in AspectC++

4.1.1 Generic Introductions by Compile-Time Introspection

Figure 6 shows the source code for a highly simplified implementation of the GOP. The keyword `aspect` in the first line declares an entity similar to a C++ class that additionally encompasses `pointcut` expressions and pieces of `advice`. A pointcut expression is a reusable alias for names defined in the program. For example, the pointcut `critical()` in line 2 lists two classes, namely `"Cyg_Scheduler"` and `"Cyg_Thread"`, from the eCos kernel. This pointcut is used by the following line that defines `advice` that those two classes get extended by a `slice` introduction, which inserts an additional member into these classes. The inserted member `"code"` is an instance of the template class `HammingCode<typename>`, whose template argument is bound to the built-in type `JoinPoint`. This type is only available in the body of advice code and offers an interface to a compile-time introspection API.

AspectC++'s introspection API [7] provides the programmer with information on the class type that is being extended by the `slice` introduction. We use this information within the template class `HammingCode` to instantiate a generative

C++ template metaprogram [14] that compiles to a tailored Hamming code for each class. In particular, we use the number of existing data members (`MEMBERS`) *prior* to the slice introduction, their types (`Member<I>::Type`) to obtain the size of each member, and a typed pointer (`Member<I>::pointer(T *obj)`) to each data member to compute the actual Hamming code. Furthermore, for classes with inheritance relationships, we recursively iterate over all base classes that are exposed by the introspection API. To simplify the iteration over this API, we implemented a Join-Point Template Library (JPTL) that offers compile-time iterators for each API entry.

4.1.2 Advice for Control Flow and Data Access

Once the Hamming code is introduced into the classes, we need to make sure that the code is checked and updated when such an object is used. At first, the Hamming code needs to be computed *whenever an object of a protected class is instantiated*. The `advice` for `construction` in line 7 implements this requirement: `after` a constructor execution, the `update()` function is invoked on the `"code"` data member. The built-in pointer `tjp->target()` yields the particular object being constructed (`tjp` is an abbreviation for **t**his **j**oin **p**oint).

The lines 11–14 define further pointcuts that describe situations where the objects are used. The pointcut function `member(...)` translates the existing pointcut `critical()` into a set of all data members and member functions belonging to classes matched by `critical()`. Thus, `call(member(critical()))` describes all procedure calls to member functions of the particular classes. Likewise, the pointcut function `get(...)` refers to all program statements that read a member variable, and the other way around, `set(...)` matches all events in the program that write to a particular member variable. The `get/set` pointcut functions are new features of the AspectC++ language that notably allow observing access to data members declared as `public`.

The `advice` in line 16 invokes the `check()` routine on the Hamming-code sub-object based on the `trigger_check()` pointcut, that is, *whenever a member function is called, or a member variable is read or written*. Similarly, the `advice` in line 20 invokes the `update()` function `after` member-function calls or writing to a member variable. Both pieces of advice invoke these routines only if the caller object (`tjp->that()`) and the callee object (`tjp->target()`) are not identical. This is an optimization that avoids unnecessary checking when an already verified object invokes a function on itself.

A call to *any* function is matched by the wild-card expression in line 25. Therewith, the advice definition in line 26 updates the Hamming code *whenever a function call leaves a critical object*, as specified by `within(member(critical()))`, and when the caller object is not identical to the callee object. When the function returns, the Hamming code gets checked by the advice in line 30.

By defining such generic pieces of advice, AspectC++ enables a modular implementation of the GOP mechanism, completely separated from the remaining

source code. More advice definitions exist in the complete GOP implementation, for instance, covering `static` data members, non-blocking synchronization, or virtual-function pointers [8].

4.2 Implementation and Evaluation

In the following, we describe the implementation of five concrete EDMs/ERMs based on the GOP mechanism. Subsequently, we demonstrate their configurability on a set of benchmark programs bundled with eCos. We show that the mechanisms can easily be adapted to protect a specific subset of the eCos-kernel data structures, e.g., only the most critical ones. After applying a heuristic that benchmark-specifically chooses this data-structure subset, and protecting the corresponding classes, we present fault injection (FI) experiment results that compare the five EDMs/ERMs. Additionally, we measure their static and dynamic overhead, and draw conclusions on the overall methodology.

4.2.1 EDM/ERM Variants

We implemented the five EDMs and ERMs listed in Table 2 to exemplarily evaluate the GOP mechanism. For instance, a template metaprogram generates an optimal Hamming code tailored for each data structure and we applied a bit-slicing technique [48] to process 32 bits in parallel. Thereby, the Hamming-code implementation can correct multi-bit errors, in particular, all burst errors up to the length of a machine word (32 bits in our case). Besides burst errors, the CRC variants (see Table 2) cover all possible 2-bit and 3-bit errors in objects smaller than 256 MiB by the CRC-32/4 code [11]. Each EDM/ERM variant is implemented as a generic module and can be configured to protect any subset of the existing C++ classes of the target system.

In the following subsections, we refer to the acronyms introduced in Table 2, and term the unprotected version of each benchmark the "Baseline."

Table 2 EDM/ERM variants, and their effective line counts (determined by `cloc`)

Variant	Description (mechanisms applied *on data member granularity*)	LOC
CRC	CRC-32, using SSE 4.2 instructions (EDM)	163
TMR	Triple modular redundancy: two copies + majority voting (EDM/ERM)	124
CRC+DMR	CRC (EDM) + one copy for error correction (ERM)	210
SUM+DMR	32-Bit two's complement addition checksum (EDM) + one copy (ERM)	198
Hamming	SW-implemented Hamming code (EDM/ERM), processing 32 bits in parallel	355
Framework	GOP infrastructure, basis for all concrete EDM/ERM implementations	2371

4.2.2 Evaluation Setup

We evaluate the five EDM/ERM variants on eCos 3.0 with a subset of the benchmark and test programs that are bundled with eCos itself, namely those 19 implemented in C++ and using threads (omitting CLOCK1 and CLOCKTRUTH due to their extremely long runtime). More details on the benchmarks can be found in previous work [8]. Because eCos currently does not support x64, all benchmarks are compiled for i386 with the GNU C++ compiler (GCC Debian 4.7.2–5), and eCos is set up with its default configuration.

Using the FAIL* FI framework [47], we simulate a fault model of uniformly distributed transient single-bit flips in data memory, i.e., we consider *all* program runs in which one bit in the data/BSS segments flips at some point in time. Bochs, the IA-32 (x86) emulator back end that FAIL* currently provides, is configured to simulate a modern 2.666 GHz x86 CPU. It simulates the CPU on a behavior level with a simplistic timing model of one instruction per cycle, also lacking a CPU cache hierarchy. Therefore the results obtained from injecting memory errors in this simulator are pessimistic, as we expect a contemporary cache hierarchy would mask some main-memory bit flips.

4.2.3 Optimizing the Generic Object Protection

As described in Sect. 4.1.1, the generic object-protection mechanisms from Table 2 can be configured by specifying the classes to be protected in a pointcut expression. Either a wild-card expression selects all classes automatically, or the pointcut expression lists a subset of classes by name. In the following, we explore the trade-off between the subset of selected classes and the runtime overhead caused by the EDM/ERMs.

We cannot evaluate all possible configurations, since there are exponentially many subsets of eCos-kernel classes—the power set. Instead, we compile each benchmark in *all* configurations that select only a single eCos-kernel class for hardening. For these sets that contain exactly one class each, we measure their simulated runtime, and subsequently order the classes from the least to most runtime overhead individually for each benchmark. This order allows us to *cumulatively* select these classes in the next step: We compile each benchmark again with increasingly more classes being protected (from one to all classes, ordered by runtime). Observing the cumulative runtimes of the respective class selections [8], the benchmarks can be divided into two categories, based on their absolute runtime:

1. **Long runtime (more than ten million cycles):** For any subset of selected classes, the runtime overhead stays negligible. The reason is that the long-running benchmarks spend a significant amount of time in calculations on the application level or contain idle phases.

2. **Short runtime (less than ten million cycles):** The EDM/ERM runtime overhead notably increases with each additional class included in the selections. These benchmarks mainly execute kernel code.

After conducting extensive FI experiments on each of the cumulatively protected programs, it turns out that for our set of benchmarks, the following heuristic yields a good trade-off between runtime and fault tolerance: *We only select a particular class if its protection incurs less than 1 percent runtime overhead*. Using this rule of thumb can massively reduce the efforts spent on choosing a good configuration, as the runtime overhead is easily measurable without running any costly FI experiments. However, in 6 of the initial 19 benchmarks, there are *no* classes that can be protected with less than 1% overhead. Those programs are most resilient without GOP (see Sect. 4.3 for further discussion).

4.2.4 Protection Effectiveness and Overhead

Using this optimization heuristic, we evaluate the EDM/ERM mechanisms described in Table 2. Omitting the aforementioned six benchmarks that our heuristic deems not protectable, Fig. 7 shows FI results from an FI campaign entailing 46 million single experiment runs, using the extrapolated absolute failure count (EAFC) as a comparison metric that is proportional to the unconditional failure probability [46]. The results indicate that the five EDM/ERMs mechanisms are similarly effective in reducing the EAFC, and reduce the failure probability by up to 79% (MBOX1 and THREAD1, protected with CRC) compared to the baseline. The total number of system failures—compared to the baseline without GOP—is reduced by 69.14% (CRC error detection), and, for example, by 68.75% (CRC+DMR error correction). Note that some benchmarks (e.g., EXCEPT1 or MQUEUE1) show very little improvement; we will discuss this phenomenon in Sect. 4.3.

Of course, the increase in system resiliency comes at different static and dynamic costs. With the GOP in place, the static binary sizes (Fig. 8) can grow quite significantly by on average 57% (CRC) to 120% (TMR) (up to 229% in the case of TMR and the KILL benchmark)—showing increases in the same order of magnitude as those observed in the *d*OSEK evaluation (Sect. 3.3.2). Looking closer, the DATA sections of all baseline binaries are negligibly tiny (around 450 bytes) and increase by 5% up to 79%. The BSS sections are significantly larger (in the tens of kilobytes), and vary more between the different benchmarks. They grow more moderately by below 1% up to 15%. In contrast, the code size (TEXT) is even larger in the baseline (23–145 kiB), and the increases vary extremely between the different variants: While CRC increases the code by an average of 114%, CRC+DMR on average adds 204%, SUM+DMR 197%, Hamming 200%, and TMR is the most expensive at an average 241% code-size increase.

But although the static code increase may seem drastic in places, low amounts of code are actually executed at runtime, as we only protected classes that introduce

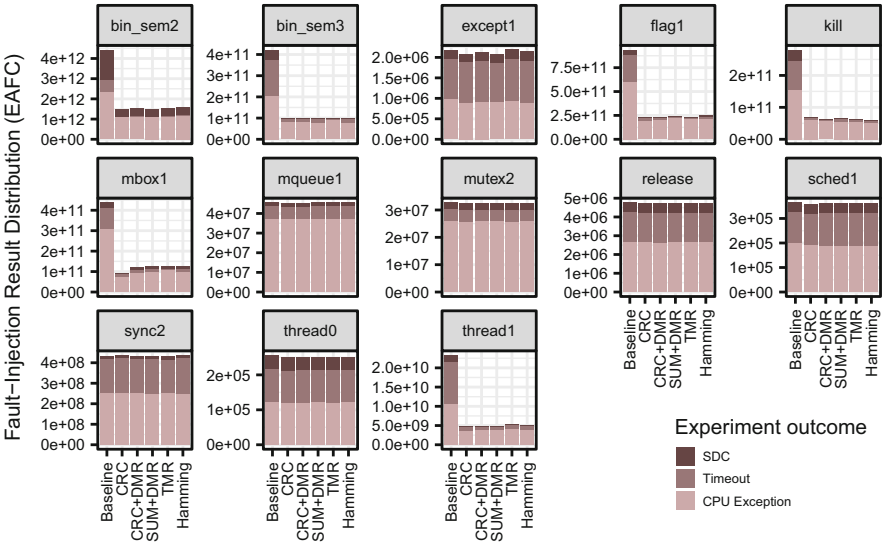


Fig. 7 Protection effectiveness for different EDM/ERM variants

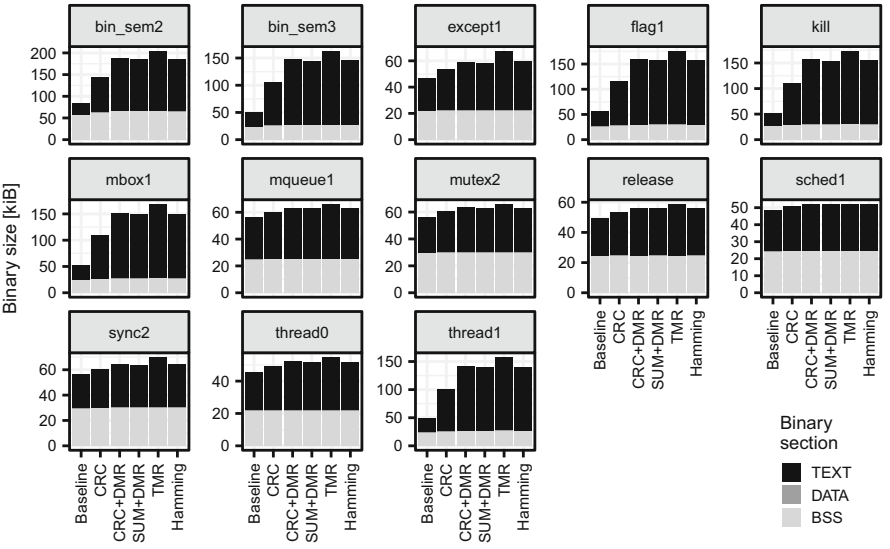


Fig. 8 Static code and data/BSS segment size of the EDM/ERM variants: the code (TEXT) segment grows due to additional CPU instructions, with CRC (detection only) being the most lightweight

less than 1% runtime overhead (see Sect. 4.2.3). Verifying the runtime on real hardware (an Intel Core i7-M620 CPU running at 2.66 GHz), we confirm that the real-world runtime overhead totals at only 0.36% for all variants except for TMR (0.37%). The results indicate that the GOP—when configured appropriately—involves negligible runtime overhead on real hardware.

4.3 Discussion

As software-implemented error detection and correction always introduces a runtime overhead, protected variants naturally run longer than their unprotected counterparts, increasing the chance of being hit by memory bit flips (assuming them to be uniformly distributed). Consequently, there exists a break-even point between, metaphorically, quickly crossing the battlefield without protection (and a high probability that a hit is fatal), and running slower but with heavy armor (and a good probability to survive a hit). The benchmarks in our initial analysis [8] we identified to be *not* effectively protectable with the GOP are on the unfavorable side of this break-even point: The additional attack surface from the runtime and memory overhead outweighs the gains from being protected for all configurations. Also, some benchmarks are just *barely* profiting from the GOP, such as, e.g., EXCEPT1 or MQUEUE1 (see Fig. 7).

A more detailed analysis of what distinguishes these benchmarks from the others reveals that they actually represent the *pathologic worst case* for GOP: Unlike "normal" applications that spend a significant amount of time in calculations on the application level, or waiting for input or events from the outside, this subset of benchmarks *only* executes eCos system calls. This reduces the time frame between an `update()` after the usage of a system object, and the `check()` at the begin of the next usage (cf. Sect. 4.1.2), to a few CPU cycles. The fault resilience gains are minimal, and the increased attack surface all in all increases the fault susceptibility significantly. Nevertheless, we do not believe the kernel-usage behavior of these benchmarks is representative for most real-world applications, and do not expect this issue to invalidate our claim that GOP is a viable solution for error detection and correction in long-living data structures.

For the remaining benchmarks, the analysis in Sect. 4.2.4 shows that the EDM/ERMs mainly differ in their static overhead. CRC is clearly the best choice when detection-only suffices. For error correction, the Hamming code turns out best. The high redundancy of the DMR variants and TMR are overkill—at least unless much more adverse fault models are considered.

5 Conserving Consistent State in Persistent Memory with Software Transactional Memory

Recent advances in persistent memory (PM) enable fast, byte-addressable main memory that maintains its state across power-cycling events. To survive power outages and prevent inconsistent application state, current approaches introduce persistent logs and require expensive cache flushes. In fact, these solutions can cause a performance penalty of up to 10× for write operations on PM. With respect to wear-out effects, and a significantly lower write performance compared to read operations, we identify this as a major flaw that impacts performance and lifetime of PM. Being already persistent, data corruptions in PM cannot be resolved by simply restarting a system. Without countermeasures this limits the usability of PM and poses a high risk of a permanently inconsistent system state.

In this section, we present ***DNV Memory***, a library for PM management. For securing allocated data against power outages, multi-bit faults that bypass hardware protection and even usage violations, *DNV Memory* introduces *reliable transactions*. Additionally, it reduces writes to PM by offloading logging operations to volatile memory, while maintaining *durability on demand* by an early detection of upcoming power failures. Our evaluation shows a median overhead of 6.5%, which is very low considering the ability to repair up to 7 random bit-errors per word. With durability on demand, the performance can be even improved by a factor of up to 3.5 compared to a state-of-the-art approach that enforces durability on each transaction commit.

5.1 System Model

We assume that hybrid system architectures equipped with both, volatile and persistent main memory, will become a commodity. This implicates that the execution state of processes will be composed of volatile and persistent parts.

While Phase Change Memory (PCM) is the most promising PM technology today, PM modules can also be built using resistive random-access memory (RRAM), spin-transfer-torque magnetoresistive random-access memory (STT-MRAM), or even battery-backed DRAM. Thereby, all processes in a system should be able to access PM directly through load and store operations in order to achieve optimal performance.

CPU caches can be used to further speed up access to persistent data. However, in order to survive power failures, cache lines containing data from PM must be flushed and the data must reach the *Durability Domain* of the PM module before the machine shuts down due to a power loss. This requires platform support in form of an asynchronous DRAM refresh (ADR) [49] or a *Flush Hint Address* [1]. Under these premises, we assume that word-level power failure atomicity is reached.

Depending on the used main-memory technology, various effects exist that may cause transient faults as previously outlined. Additionally, PCM and RRAM have

a limited write endurance that lies in the range of 10^6 up to 10^{10} operations [31]. Once worn out, the cell's value can only be read but not modified anymore.

We assume that all static random-access memory (SRAM) cells inside the CPU are guarded by hardware fault tolerance and are sufficiently reliable to ensure correct operation. Of course reliable DRAM supporting hardware error correction code (ECC) exists and PM can be protected by hardware solutions too. However, the common hardware ECC mechanisms only provide single-bit-error correction, double-bit-error detection (SECDED) capabilities, which is not always sufficient [52]. We assume that due to economic reasons not every PM module will support the highest possible dependability standard, leaving a fraction of errors undetected. Some PM modules may even lack any hardware protection. This paves the way for software-based dependability solutions.

5.2 Concepts of DNV Memory

The main goal of our design is to provide the familiar *malloc* interface to application developers for direct access to PM. At the same time, we want data stored in PM to be robust against power failures, transient faults, and usage errors.

Our core API functions (see Table 3(a) and (b)) resemble the interface of `malloc` and `free`. The only additional requirement for making legacy volatile structures persistent with *DNV Memory* is using our API functions and wrapping all persistent memory accesses in atomic blocks (see Table 3(e)).

These atomic blocks provide ACID[3] guarantees for thread safety, and additionally preserve consistency in case of power failures. Furthermore, *DNV Memory* combines software transactional memory (STM) with the allocator to manage

Table 3 Overview of the *DNV Memory* application programming interface (API)

Category	Function	Description	Ref.
Core API	void* dnv_malloc(size_t sz)	Allocates persistent memory like **malloc(3)**	(a)
	void dnv_free(void* ptr)	Releases persistent memory like **free(3)**	(b)
Static Variables	DNV_POD *variable*	Statically places *plain old data* in PM at definition	(c)
	DNV_OBJ *variable*	Statically places the object in PM at definition	(d)
Transactions	__transaction_atomic{...}	Atomic block with ACID guarantees and reliability	(e)

[3] Atomicity, consistency, isolation, durability.

software-based ECC. Every data word that is accessed during a transaction is validated and can be repaired if necessary.

In order to store entry points to persistent data structures that survive process restarts, *DNV Memory* provides the possibility to create static persistent variables (Table 3(c) and (d)). On top of this core functionality, *DNV Memory* introduces the concepts *durability on demand* and *reliable transactions* that are explained in the following.

If a power failure occurs during the update of persistent data structures, the *DNV Memory* might be in an inconsistent state after restart. To prevent this, *DNV Memory* follows the best practices from databases and other PM allocators [12, 54] and wraps operations on PM in atomic blocks. This can be achieved with STM provided by modern compilers or libraries like TinySTM [22]. The transactions must also be applied to the allocator itself, as its internal state must be stored in PM as well.

Different to previous works, *DNV Memory* aims at minimizing write accesses to PM. We store all transaction logs in volatile memory and utilize a power-failure detection to enforce *durability on demand*. When a power outage is imminent, the operating system copies the write-back logs back to PM in order to prevent state inconsistency. Therefore, every thread has to register its volatile memory range for the write-back log at our kernel module, which in turn reserves a PM range for a potential backup copy. After restart, the write-back logs are restored from PM, and every unfinished commit is repeated.

Since durability is actually required only in case of a power failure or process termination, memory fences and cache flushing can be performed on demand. This preserves persistent data inside the CPU cache and consequently reduces writes to PM. Additionally, since memory within a CPU is well protected by hardware, persistent data inside the cache is less susceptible to transient faults and can be accessed faster.

Enforcing durability on demand requires the ability to detect power failures in advance. For embedded devices, the power-outage detection is a part of the *brownout* detection and state of the art [43]. On servers and personal computers, power outages can be detected via the PWR_OK signal according to the ATX power supply unit (PSU) design guide [3]. Although the PWR_OK signal is required to announce a power outage at least 1 ms in advance, much better forecasts can be achieved in practice. For instance, some Intel machines provide a power-failure forecast of up to 33 ms [39]. An even better power-failure detection can be achieved by inspecting the input voltage of the PSU with a simple custom hardware [25]. With this approach, power failures can be detected more than 70 ms in advance, which leaves more than enough time to enforce durability and prevent further modification of persistent data.

Crashes that are not caused by power failures can be handled just like power failures if durability can be secured. For instance, our kernel module is aware of any process using PM that terminates and enforces durability in that case. Crashes in the operating-system kernel can be handled either as part of a kernel-panic procedure, or by utilizing a system like Otherworld [16].

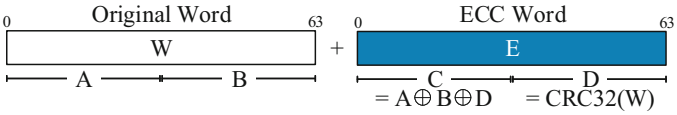


Fig. 9 *DNV Memory* ECC

In order to protect persistent data from corruption, *DNV Memory* reserves additional memory in each allocation that is meant to store ECC data. Afterwards fault tolerance is provided through *reliable transactions*.

As described in the previous section, all accesses to PM should be wrapped by atomic blocks in order to protect persistent data from power failures. These atomic blocks simply wrap all read and write operations in `TM_LOAD` and `TM_STORE` functions provided by the STM library, which in consequence control every word access. In combination with support from the memory allocator, this can be exploited to provide transparent fault tolerance.

Essentially, any ECC can be used to provide fault tolerance in software. For instance, we considered the SECDED Hamming code that is common in hardware protected memory. It protects 64-bit words with additional 8 bits, resulting in a 12.5% memory overhead. However, if implemented in software, the Hamming code would highly impact the performance of the application. Additionally, as already mentioned, we do not think that SECDED is enough to protect persistent data. Consequently, we decided to implement an ECC that provides a high multi-bit error correction with a memory overhead no more than dual modular redundancy. In addition, we want a fast error detection in software by exploiting commonly available hardware support. In general, whenever a *data word* W is written inside an atomic block, an *ECC word* E is created and stored in the additional space that the allocator has reserved. In theory, any fault-tolerant encoding is possible as long as error detection can be conducted in a few CPU cycles.

For *DNV Memory* we combine cyclic redundancy check (CRC) for fast error detection with an error location hint. Thus, we subdivide E into two halves C and D as shown in Fig. 9. The *error detection* half word D is generated with CRC32c ($D = CRC32c(W)$). We chose CRC as hardware support is available on many architectures, including most commodity CPUs. Additionally, with CRC32c—which is supported by *SSE 4.2*,—a Hamming distance of 8 is achieved on a word length of 64 bits [33]. Without further assistance, error correction of up to 3 bits can be achieved by guessing the error location. However, by augmenting the CRC-based error detection with an *error location hint* C, less trials are needed and more bit-errors can be corrected. Inspired by RAID level 5 [42], we subdivide the data word W into two halves A and B and compute C according to Eq. (1).

$$C = A \oplus B \oplus D \tag{1}$$

The data validation takes place during a transaction whenever a word W is read for the first time. At that point, we recompute E' from W and compare its value with E. Normal execution can continue if both values match. Otherwise error correction is initiated.

Since errors can be randomly distributed across W and E, we start the error correction by narrowing the possible locations of errors. Therefore, we compute the *error vector* F via Eq. (2), which indicates the bit position of errors.

$$F = A \oplus B \oplus C \oplus D \tag{2}$$

This information is, however, imprecise, as it is unknown whether the corrupted bit is located in A, B, C, or D. Thus, for f *errors detected by* F, 4^f repair candidates R_i are possible, and are computed via Eq. (3). The *masking vectors* M_a, M_b, M_c, M_d are used to partition F between all four half words.

$$\begin{aligned} R_i &= W_i \| E_i \\ W_i &= A \oplus (F \wedge M_a) \| B \oplus (F \wedge M_b) \\ E_i &= C \oplus (F \wedge M_c) \| D \oplus (F \wedge M_d) \end{aligned} \tag{3}$$

To find the repair candidate R_s that contains the right solution, each R_i needs to be validated by recomputing E_i' from W_i and compare it to E_i. In order to repair all errors, exactly one R_s must be found with matching E_i' and E_i. For instance, if all errors are located in A, the repair candidate using $M_a = F$ and other masking vectors set to zero will be the correct result. Additionally, all combinations need to be considered that have an error at the same bit position in two or all half words, as these errors extinguish each other in C.

Please note that the set of repair candidates may yield more than one solution that can be successfully validated if more than three errors are present. To prevent a false recovery, all repair candidates must be validated for up to n errors. As an optimization step, we estimate n by counting the population in $E \oplus E'$ and limit the result to a maximum of $n = 7$.

To optimize the performance in a cache-aware way, we store the ECC words interleaved with the original words W as presented in Fig. 10. However, this interleaved data layout cannot be accessed correctly outside atomic blocks because the original layout is always expected here. Unfortunately, omitting atomic blocks around PM access is a very common mistake. We encountered such usage errors in every single STAMP benchmark [38], and whenever we ported or wrote persistent applications ourselves. Since the access to PM outside atomic blocks should be prevented to keep data consistent during power failures, we introduce the concept of a *transaction staging (TxStaging) section* as shown in Fig. 10. All memory that is allocated by *DNV Memory* has addresses belonging to the TxStaging section. The same applies to the location of persistent static variables. The TxStaging section is only a reserved virtual address space without any access rights. Consequently, any access to this segment will cause a segmentation fault that is easy to debug.

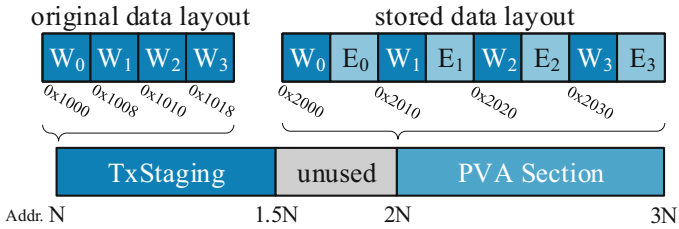


Fig. 10 *DNV Memory* persistent data layout and memory sections

However, inside an atomic block every access to the TxStaging section is intercepted by the STM library and redirected to the persistent virtual address (PVA) section where the actual persistent data is stored. To simplify the address transformation, the PVA section should be located at the address of the TxStaging section multiplied by 2. For instance, assuming the TxStaging section begins at address `0x1000` the PVA section should be placed at `0x2000`. In that case a 32-byte object that is located in the address range from `0x1000` to `0x101f` will be transformed into the address space `0x2000` to `0x203f` as shown in Fig. 10.

5.3 Evaluation

We implemented *DNV Memory* on Linux in the form of a user-space library with a small companion kernel module and a hardware power-failure detector. Our design does not require any changes to the operating-system kernel or the machine itself. All components are pluggable and can be replaced by more extended solutions if needed. All user-space code is written in C++ and compiled with an unmodified GCC 5.4.0. A small linker-script extension provides additional sections like the TxStaging or the PVA section as shown in Fig. 10.

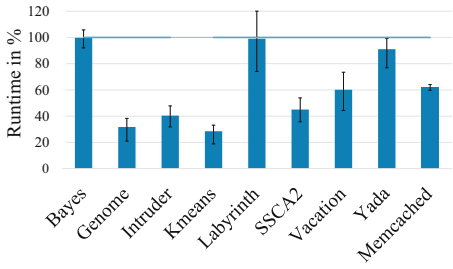
To show the feasibility of *durability on demand*, we artificially introduced power failures and measured the time between the detection of a power failure and the eventual machine shutdown. This period is referred as the shutdown forecast, and the results of 100 experiments are shown in Fig. 11. Additionally, the time of critical tasks in the event of a power failure is shown here. As can be seen, power failures can be detected sufficiently early to conduct all necessary durability measures. Counterintuitively, an idling CPU has a negative impact on the feasibility of the approach because the CPU enters the a power-saving mode with reduced performance. Additionally, less energy is stored within the power supply in the event of a power failure, thus leading to a quicker shutdown.

The performance impact of durability on demand was evaluated with applications from the STAMP benchmark suite [38] and the Memcached key-value store that was retrofitted with transactions. Figure 12 shows for each application the average relative runtime out of 100 measurements together with the 90% quantile that is

Fig. 11 Duration of critical tasks. A Heavy workload is achieved through kernel compilation

Measurement	Workload	Time in ms min	Time in ms max
Stop CPU and Flush Cache	Heavy	2.3	3.3
	Idle	4.4	5.6
Store Write-Back Log	Heavy	3.8	4.8
	Idle	7.4	8.6
Shutdown Forecast	Heavy	34.6	39.4
	Idle	25.2	36.8

Fig. 12 Application runtime under *durability on demand* in comparison to *durability on commit* (100% baseline)



indicated by the error bars. As the 100% baseline we used the state of the art, which enforces durability on each transaction commit. The results highly correlate with the cache efficiency of the application. For instance, little to no performance impact was achieved for Bayes, Labyrinth, and Yada, which operate on large work sets and show large transactions. If the transactions become large, they do not fit well into the cache and therefore do not benefit from locality, which severely impacts performance. Enforcing durability in this case has a low impact because the overhead from memory barriers and cache flushing becomes negligible. The other benchmarks, however, have moderate to small work sets, therefore a significant performance increase of up to 3.5× can be observed.

To investigate the error detecting and correcting capabilities of *DNV Memory*, we conducted one billion fault-injection experiments, for one to seven-bit errors each. Every fault-injection experiment used a random word and bit-error positions that were randomly distributed over the original data and its corresponding ECC word. Only in the case of 7-bit errors, a small fraction of 0.000012163% fault injections produced ambiguous repair solutions that prevented a correction. In all other cases, including all errors up to 6-bit, a detection and correction was always successful. As can be seen in Fig. 13 the repair time increases exponentially with the number of flipped bits. However, even for correcting seven-bit errors, the mean error-repair time is less than 1.4 ms, which is acceptable considering the low probability of errors. Without any error, the validation only takes 34 ns.

For the performance evaluation of reliable transactions we again used STAMP benchmark applications [38] and Memcached. The bars depicted in Fig. 14 show

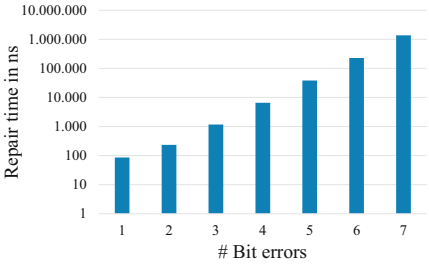


Fig. 13 Time to repair bit errors with reliable transactions

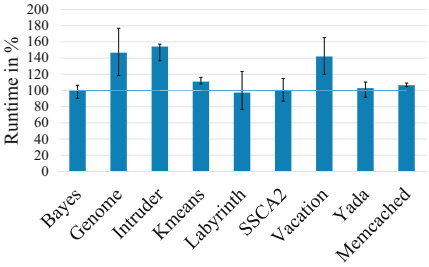


Fig. 14 Performance impact of reliable vs. traditional transactions (100% baseline)

the mean runtime of each benchmark. All values are relative to plain transactional execution (the 100% baseline), and the error bars represent the 95% and the 5% quantile. Over all applications, a median runtime of 106.5% is achieved with reliable transactions. Applications above this median have a workload that is dominated by reads or short transactions, hence the overhead of data verification has a higher impact here. Applications with a balanced or write-driven workload, however, have a higher runtime impact from transactions in general, thus the overhead that comes from reliable transactions is less prevalent. In summary, these results indicate a very acceptable performance impact—especially when considering the error-correcting capabilities of the approach.

5.4 Discussion

DNV Memory provides system support for dependable PM. Unlike previous approaches, *DNV Memory* enforces *durability on demand*, which in turn reduces write operations on PM and therefore improves reliability, lifetime, and performance. For tolerating power failures, *DNV Memory* uses software transactions that also include and secure the allocator itself. Our system even goes one step further and provides fault tolerance via software transactional memory. As our evaluation showed, *DNV Memory* protects data at word granularity, with an ECC word that is capable of detecting and correcting *a random distributed seven-bit error*, which is by far more than common hardware protection offered by server-

class volatile main memory. We also demonstrated that power failures can be detected early, allowing to conduct all necessary cleanup operations.

6 Summary

The work presented in this chapter has gained high visibility in the international research community. It was on the programme of all major conferences in the field and the authors received a number of best paper, best poster, and best dissertation awards, culminating in the renowned Carter Award for Christoph Borchert.

A reason for this success might be the focus on design principles and methods for hardening the operating system—and only the operating system. Most of previous research did not consider the specific properties of this special execution environment, such as different kinds of concurrent control flows, or assumed the reliable availability of underlying system services.

In our work we made a huge effort to design and implement an embedded operating system from scratch with the goal to explore the limits of software-implemented hardware fault tolerance in a reliability-oriented static system design. As a result we were able to reduce the SDC probability by orders of magnitude and found the remaining spots where software is unable to deal with hardware faults.

For existing embedded operating systems we have developed and evaluated Generic Object Protection by means of "dependability aspects," which can harden operating systems at low cost without having to change the source code, and also addressed faults that crash the whole system by means of reliable transactions on persistent memory.

Finally, the authors have developed a fault-injection framework for their evaluation purposes that implements novel methods, which also advanced the state of the art in this domain.

Acknowledgments This work was supported by the German Research Foundation (DFG) under priority program SPP-1500 grants no. KA 3171/2-3, LO 1719/1-3, and SP 968/5-3.

References

1. Advanced Configuration and Power Interface Specification (Version 6.1) (2016). http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf
2. Alexandersson, R., Karlsson, J.: Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In: Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11), pp. 303–314. IEEE Press, Piscataway (2011). https://doi.org/10.1109/DSN.2011.5958244
3. ATX12V Power Supply Design Guide (2005). http://formfactors.org/developer%5Cspecs%5CATX12V_PSDG_2_2_public_br2.pdf

4. Aussagues, C., Chabrol, D., David, V., Roux, D., Willey, N., Tornadre, A., Graniou, M.: PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects. In: Proceedings of the 4th International Conference on Embedded Real Time Software and Systems (ERTS2 '10) (2010)
5. AUTOSAR: Specification of operating system (version 5.1.0). Tech. rep., Automotive Open System Architecture GbR (2013)
6. Bhandari, K., Chakrabarti, D.R., Boehm, H.J.: Makalu: fast recoverable allocation of non-volatile memory. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2016)
7. Borchert, C., Spinczyk, O.: Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In: Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15), pp. 1–7. ACM Press, New York (2015). https://doi.org/10.1145/2818302.2818304
8. Borchert, C., Schirmeier, H., Spinczyk, O.: Generic soft-error detection and correction for concurrent data structures. IEEE Trans. Dependable Secure Comput. **14**(1), 22–36 (2017). https://doi.org/10.1109/TDSC.2015.2427832
9. Borkar, S.Y.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6), 10–16 (2005). https://doi.org/10.1109/MM.2005.110
10. Cassens, B., Martens, A., Kapitza, R.: The neverending runtime: using new technologies for ultra-low power applications with an unlimited runtime. In: International Conference on Embedded Wireless Systems and Networks, NextMote Workshop (EWSN 2016) (2016)
11. Castagnoli, G., Brauer, S., Herrmann, M.: Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. IEEE Trans. Commun. **41**(6), 883–892 (1993). https://doi.org/10.1109/26.231911
12. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: SIGARCH Computer Architecture News (2011). https://doi.org/10.1145/1961295.1950380
13. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the Symposium on Operating Systems Principles (2009)
14. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Boston (2000)
15. Danner, D., Müller, R., Schröder-Preikschat, W., Hofer, W., Lohmann, D.: Safer Sloth: efficient, hardware-tailored memory protection. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '14), pp. 37–47. IEEE Press, Piscataway (2014)
16. Depoutovitch, A., Stumm, M.: "Otherworld" - giving applications a chance to survive OS kernel crashes. In: Proceedings of the European Conference on Computer Systems (EuroSys) (2010)
17. Dietrich, C., Hoffmann, M., Lohmann, D.: Cross-kernel control-flow-graph analysis for event-driven real-time systems. In: Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15). ACM Press, New York (2015). https://doi.org/10.1145/2670529.2754963
18. Dietrich, C., Hoffmann, M., Lohmann, D.: Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. ACM Trans. Embed. Comput. Syst. **16**, 35:1–35:25 (2017). https://doi.org/10.1145/2950053
19. Döbel, B., Härtig, H.: Who watches the watchmen?—protecting operating system reliability mechanisms. In: International Workshop on Hot Topics in System Dependability (HotDep) (2012)
20. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the 9th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '14) (2014)
21. ERIKA Enterprise. https://erika.tuxfamily.org. Accessed 29 Sept 2014

22. Felber, P., Fetzer, C., Riegel, T., Marlier, P.: Time-based software transactional memory. IEEE Trans. Parallel Distrib. Syst. **21** (2010). https://doi.org/10.1109/TPDS.2010.49
23. Forin, P.: Vital coded microprocessor principles and application for various transit systems. In: Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation (CCCT '89), pp. 79–84 (1989)
24. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Software-Implemented Hardware Fault Tolerance. Springer, New York (2006). https://doi.org/10.1007/0-387-32937-4
25. Heiser, G., Le Sueur, E., Danis, A., Budzynowski, A., Salomie, T.l., Alonso, G.: RapiLog: reducing system complexity through verification. In: Proceedings of the 8th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '13) (2013). https://doi.org/10.1145/2465351.2465383
26. 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., Teich, J., Wehn, N., Wunderlich, H.J.: Design and architectures for dependable embedded systems. In: Dick, R.P., Madsen, J. (eds.) Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11), pp. 69–78. ACM Press (2011). https://doi.org/10.1145/2039370.2039384
27. Hoffmann, M., Borchert, C., Dietrich, C., Schirmeier, H., Kapitza, R., Spinczyk, O., Lohmann, D.: Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In: Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14), pp. 230–237. IEEE Press, Piscataway (2014). https://doi.org/10.1109/ISORC.2014.26
28. Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., Schröder-Preikschat, W.: A practitioner's guide to software-based soft-error mitigation using AN-codes. In: Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14), pp. 33–40. IEEE Press, Miami (2014). https://doi.org/10.1109/HASE.2014.14
29. Hoffmann, M., Lukas, F., Dietrich, C., Lohmann, D.: dOSEK: the design and implementation of a dependability-oriented static embedded kernel. In: Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '15), pp. 259–270. IEEE Press, Piscataway (2015). https://doi.org/10.1109/RTAS.2015.7108449
30. IEC: IEC 61508 – functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission, Geneva (1998)
31. Kannan, S., Gavrilovska, A., Schwan, K.: pVM: persistent virtual memory for efficient capacity scaling and object storage. In: Proceedings of the 11th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '16), pp. 13:1–13:16. ACM, New York (2016). https://doi.org/10.1145/2901318.2901325
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97). Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer, Berlin (1997). https://doi.org/10.1007/BFb0053381
33. Koopman, P.: 32-Bit cyclic redundancy codes for Internet applications. In: Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02) (2002). https://doi.org/10.1109/DSN.2002.1028931
34. Li, Y., West, R., Missimer, E.: A virtualized separation kernel for mixed criticality systems. In: Proceedings of the 10th USENIX International Conference on Virtual Execution Environments (VEE '14), pp. 201–212. ACM Press, New York (2014). https://doi.org/10.1145/2576195.2576206
35. Mariani, R., Fuhrmann, P., Vittorelli, B.: Fault-robust microcontrollers for automotive applications. In: Proceedings of the 12th International On-Line Testing Symposium (IOLTS '06), 6 pp. IEEE Press, Piscataway (2006). https://doi.org/10.1109/IOLTS.2006.38
36. Martens, A., Scholz, R., Lindow, P., Lehnfeld, N., Kastner, M.A., Kapitza, R.: Dependable non-volatile memory. In: Proceedings of the 11th ACM International Systems and Storage

Conference, SYSTOR '18, pp. 1–12. ACM Press, New York (2018). https://doi.org/10.1145/3211890.3211898
37. Massa, A.: Embedded Software Development with eCos. Prentice Hall, Upper Saddle River (2002)
38. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: Proceedings of the International Symposium on Workload Characterization (IISWC) (2008)
39. Narayanan, D., Hodson, O.: Whole-System Persistence, pp. 401–410. ACM Press, New York (2012). https://doi.org/10.1145/2189750.2151018
40. OSEK/VDX Group: Operating system specification 2.2.3. Tech. rep., OSEK/VDX Group (2005). http://portal.osek-vdx.org/files/pdf/specs/os223.pdf. Accessed 29 Sept 2014
41. Pattabiraman, K., Grover, V., Zorn, B.G.: Samurai: protecting critical data in unsafe languages. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08), pp. 219–232. ACM Press, New York (2008). https://doi.org/10.1145/1352592.1352616
42. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). SIGMOD Rec. **17**(3), 109–116 (1988). https://doi.org/10.1145/971701.50214
43. Philips Semiconductors: AN468: Protecting Microcontrollers against Power Supply Imperfections (2001)
44. Rebaudengo, M., Sonza Reorda, M., Violante, M., Torchiano, M.: A source-to-source compiler for generating dependable software. In: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, pp. 33–42. IEEE Press (2001). https://doi.org/10.1109/SCAM.2001.972664
45. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-encoding: detecting hardware errors in software. In: Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP '10), pp. 169–182. Springer, Berlin (2010)
46. Schirmeier, H., Borchert, C., Spinczyk, O.: Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In: Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15), pp. 319–330. IEEE Press, Piscataway (2015). https://doi.org/10.1109/DSN.2015.44
47. Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., Spinczyk, O.: FAIL*: an open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In: Proceedings of the 11th European Dependable Computing Conference (EDCC '15), pp. 245–255. IEEE Press, Piscataway (2015). https://doi.org/10.1109/EDCC.2015.28
48. Shirvani, P.P., Saxena, N.R., McCluskey, E.J.: Software-implemented EDAC protection against SEUs. IEEE Trans. Reliab. **49**(3), 273–284 (2000). https://doi.org/10.1109/24.914544
49. SNIA NVDIMM Messaging and FAQ (2014)
50. Song, J., Wittrock, J., Parmer, G.: Predictable, efficient system-level fault tolerance in C^3. In: Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS '13), pp. 21–32. IEEE Press (2013). https://doi.org/10.1109/RTSS.2013.11
51. Spinczyk, O., Lohmann, D.: The design and implementation of AspectC++. Knowl.-Based Syst. **20**(7), 636–651 (2007). Special Issue on Techniques to Produce Intelligent Secure Software. https://doi.org/10.1016/j.knosys.2007.05.004
52. Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K.B., Stearley, J., Shalf, J., Gurumurthi, S.: Memory errors in modern systems: the good, the bad, and the ugly. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15), pp. 297–310. ACM Press, New York (2015). https://doi.org/10.1145/2694344.2694348
53. Ulbrich, P., Hoffmann, M., Kapitza, R., Lohmann, D., Schröder-Preikschat, W., Schmid, R.: Eliminating single points of failure in software-based redundancy. In: Proceedings of the 9th European Dependable Computing Conference (EDCC '12), pp. 49–60. IEEE Press, Piscataway (2012). https://doi.org/10.1109/EDCC.2012.21

54. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: SIGARCH Computer Architecture News, vol. 39, pp. 91–104. ACM Press, New York (2011). https://doi.org/10.1145/1961295.1950379
55. Yeh, Y.C.: Triple-triple redundant 777 primary flight computer. In: Proceedings of the IEEE Aerospace Applications Conference, vol. 1, pp. 293–307 (1996). https://doi.org/10.1109/AERO.1996.495891

Part II
Cross-Layer Dependability: From Architecture to Software and Operating System

Michael Engel

Designers of modern embedded systems have to cope with errors in all kinds of system components, such as processing elements, memories, I/O devices, and interconnects. The ever increasing pressure to reduce the size, cost, and energy consumption of a given system has two effects that tend to amplify each other. On the one hand, solutions that are able to mitigate errors on the hardware side are often considered too expensive in terms of product cost or energy consumption and, thus, are frequently left out of the design for not strictly safety-critical systems. On the other hand, the ongoing miniaturization of semiconductor feature sizes and the reduction of supply voltages results in hardware that is increasingly more sensitive to external effects, such as cosmic radiation, thermal effects, or electromagnetic interference, that could cause errors.

As a consequence, those errors are much more likely to affect recent and future designs. The design constraints, thus, require new methods to detect and mitigate errors and allow designers to create more cost- and energy-efficient systems.

Due to the wide spectrum of possible error causes, approaches to mitigate these errors vary significantly. In this book section, a number of approaches that have been developed in the context of SPP 1500 as well as in projects of collaborating researchers are presented that cover a large part of the possible design space. The different projects discussed in the following chapters have one important common property—they are not restricted to work on a single layer of the hardware/software stack, but instead integrate information from different layers of the stack for increased efficiency.

The first chapter of this section, written by Kühn et al., analyzes the opportunities that massively parallel architectures offer in terms of providing a platform for the reliable execution of software. A large number of available processors enable the system designer to make use of selective redundancy for differing requirements

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

of the system's software components. The use of online health monitoring allows the system to detect typical reliability issues such as negative-bias temperature instability (NBTI) or hot carrier injection (HCI) and adapt to these by using online hardware reconfiguration before the issues result in an error visible on the software layers.

The second chapter by Kriebel et al. tries to tackle the reliability problem from a different direction. In their approach, the authors use error models and additional information from hardware as well as software layers in order to generate dependable software. By quantifying the error masking and propagation properties of a system, an analysis is performed that determines in which way an application's output will be affected by the assumed errors. An increase in dependability is then achieved by avoiding or mitigating the critical situations by means of software transformation or selective instruction protection.

In the third chapter by Kriebel et al., an approach to protect systems against transient errors using heterogeneous hardware/software hardening is proposed. Here, the authors analyze and exploit masking and error tolerance properties of different levels of the hard- and software stack. By using system components with different reliability properties from the architecture level to the design of caches, systems are enabled to adapt to error properties and reliability requirements of the executed software. The authors also give an outlook onto methods to complement the described heterogeneous hardware approach with compiler-based heterogeneous hardening modes on the software level.

The fourth chapter by van Santen et al. concentrates on reliability optimization for embedded multiprocessor systems on chip (MPSoCs). The problems analyzed are interdependencies of temperature and the reliable operation of MPSoCs. Here, the authors employ measured or estimated thermal values for different cores to determine which measures on system level can be applied to balance the thermal stress. This balancing, in turn, results in an evenly distributed probability of errors throughout the system. To enable the balancing, task migration between different cores based on virtualized interconnects is employed, which enables fast and transparent switch-over of communication channels.

Memory errors are in the focus of the fifth chapter, contributed by Alam and Gupta. The optimization of current memory chips to maximize their bit storage density makes them especially susceptible to soft errors. For cost and efficiency reasons, this process neglects to optimize for additional parameters such as manufacturing process variation, environmental operating conditions, and aging-induced wearout, leading to significant variability in the error susceptibility of memories. To improve memory reliability, the authors propose to replace traditional hardware-based bit-error checking and correction methods by software managed techniques and novel error correction codes to opportunistically cope with memory errors. These techniques can take the architectural or application context into consideration by leveraging semantics information to reduce the cost of error correction.

The final chapter in this section by Ma et al. concentrates on MPSoC systems again. Here, the focus lies on the contrasting requirements of soft-error reliability and lifetime reliability. The authors observe that most existing work on MPSoC fault tolerance only considers one of the described requirements, which in turn

might adversely impact the other. Accordingly, the possible tradeoffs between soft-error reliability and lifetime reliability are analyzed in order to achieve a high overall system reliability. Like some of the previously described approaches, the authors make efficient use of heterogeneous MPSoC architecture properties, such as big-little type same-ISA systems and systems that integrate traditional CPUs with GPGPUs.

Overall, the different approaches discussed in this section cover a large part of typical modern embedded architectures. Solutions for improved reliability of processors, memories, and interconnects are presented. A common theme for all these approaches is that each one operates on several different layers of the hardware/software stack in order to exploit this fused information to reduce the hardware and software overhead for error detection and mitigation.

While this common property is shown to be beneficial for embedded design tasks facing dependability problems, the particular projects show a large variety of detail in their approaches to achieve that goal. One common approach is to operate in a bottom-up way. These systems adapt hardware properties to mask problems for the software level. Other approaches make use of a top-down methodology. Here, software is adapted in order to handle possible errors showing up in the hardware. In general, however, most of the projects described above employ sort of a hybrid approach, in which information from different layers of the system is fused in order to enable optimization decisions at compile time and runtime.

An important additional research direction reflected in this section is based on the idea of accepting certain incorrect behaviors of a system in response to an error. Here, additional semantic information on the relevance of deviating system behavior is employed to determine the criticality of certain errors. In turn, accepting certain imprecisions in a system's results enables more efficient reliable embedded systems.

In general, we can conclude that all of the cross-layer techniques described above show significant improvements in the non-functional properties or design constraints a system designer has to consider when creating dependable embedded systems.

The large variety of analysis and mitigation efforts throughout all layers of the hardware and software stack show that dependability of systems, even if it is one of the earliest research topics in computer engineering, is still a highly relevant and active research topic. Novel challenges due to different hardware components and their properties increased demands on the computational power and energy efficiency as well as additional non-functional properties require innovative methods that combine work on all layers of the hardware and software stack.

However, we have to assess that the current solution landscape, of which we have tried to show a representative profile in this section, today still tends to produce isolated solutions which are not designed for interoperability. Here, an important future research challenge is an overarching effort that allows to flexibly integrate information from various different layers and, in turn, to enable multi-criterial optimizations at design and compile time as well as at runtime to enable general fault tolerant embedded systems. It will be interesting to observe how the different approaches described in this section will be able to contribute to this overall objective.

Increasing Reliability Using Adaptive Cross-Layer Techniques in DRPs: Just-Safe-Enough Responses to Reliability Threats

Johannes Maximilian Kühn, Oliver Bringmann, and Wolfgang Rosenstiel

1 Introduction

The broad deployment, as well as the increasingly difficult manufacturing of in-spec semiconductors long make reliable operation and failures across the lifetime of an embedded system one of the industry's main concerns. Since ever-increasing demands do no longer allow us to resort to "robust" technologies, other means than semiconductor technology have to fill the gap left by cutting-edge technologies without resorting to unrealistic mainframe like protection mechanisms. As the operation scenarios become ever more challenging as well (edge computing, intelligent IoT nodes), hardware architects are faced with ever tighter power budgets for continuously increasing compute demands. We, therefore, proposed to exploit the architectural redundancies provided by potent, yet energy efficient massively parallel architectures, modeled using Dynamically Reconfigurable Processors (DRP). Using DRPs, we built an extensive cross-layer approach inspired by the overall project's approach as laid out in [1]. Following the idea of cross-layer reliability approaches, we built interfaces reaching from software layers right down to the transistor level mainly through computer architecture, allowing us to address both the varying reliability requirements and the significant computational demands of prospective workloads.

Figure 1 shows an overview of the layers this project targeted as described in the previous paragraph. While a strong focus has been on architecture, the project's aim was to use computer architecture to connect to the layers above and

J. M. Kühn (✉)
Preferred Networks Inc., Tokyo, Japan
e-mail: kuhn@preferred.jp

O. Bringmann · W. Rosenstiel
Eberhard Karls Universität Tübingen, Tübingen, Germany
e-mail: oliver.bringmann@uni-tuebingen.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_5

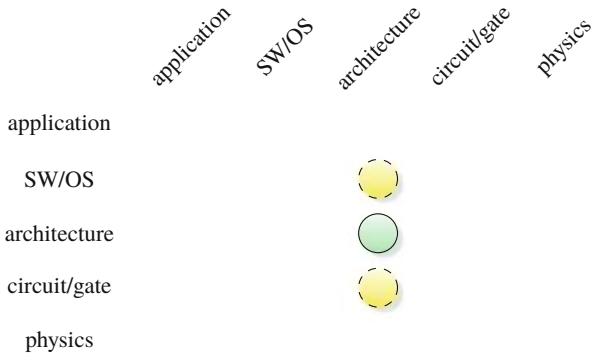


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

below. We show how DRP architectures can leverage their inherent architectural redundancies to realize various degrees of reliable computing. On one end of the spectrum, we highlight how triple modular redundancy (TMR) and duplication with comparison (DWC) compute modes can be realized to actively secure computations without permanently binding hardware resources and with only slight hardware overheads. On the other end of the spectrum, we show how fault-free operation can be passively ascertained by periodically testing SoC components. Both, active and passive concepts together with the architectural redundancies allow for graceful degradation by pinpoint failure detection and subsequently dynamically remapping applications. Once established, both graceful degradation and low-cost TMR for critical parts of applications can be used to make specific operations in processor cores reliable by using the DRP or the demonstrated concepts as a reliable pipeline within a processor core.

A central point of the proposed methods is an overarching cross-layer approach [1], tying together these methods from the software layers (Application, Operating System) to all hardware layers below down to the semiconductor through the concepts introduced by our DRP architecture. To enable a reach down to the circuit level, we exemplarily used the extensive Body Biasing capabilities of Fully Depleted Silicon on Insulator (FDSOI) processes as a means for transistor-level testing and manipulation. This access down to the transistor level enables continuous monitoring of the precise hardware health and thereby not only reactive measures in case of hardware failure but also proactive measures to prevent system failure and prolong system lifetime if the hardware starts exhibiting signs of wear. Access to the device state also multiplies the reliability and system health options on the software layer. With previously having the choice of using TMR/DWC to minimize the error probability, we also show how DVFS with Body Biasing can offer both high power but highly reliable over spec versus ultra-low-power but risky computing modes. These modes' long-term effects further multiply the set of operation modes, e.g., slowing down or speeding up degenerative effects such as Hot

Carrier Injection (HCI) or Negative Bias Temperature Instability (NBTI). However, with access to actual transistor parameters, the proposed approach also indicates that even permanent degeneration such as HCI can be temporarily overcome [2] to prolong system lifetime long enough to extend the graceful degradation period beyond conventional physical limits. Or to put it in the spirit of the parallel NSF effort [3], by opportunistically filling the technology gap using cross-layer methods, there are more means to approach and exploit the hardware's sheer physical limits.

Within this project, we also faced the challenge of how such cross-layer approaches can be realistically validated and evaluated. While Software layers down to the RTL level allow, e.g., fault injection through instrumentation or emulation, the computational effort quickly becomes too large for realistically sized test samples. Furthermore, going below the gate level offers an entirely new set of challenges, both calling for appropriate solutions. For the layers from Software to RTL, we chose to implement the entire system as a prototype on an FPGA. For this FPGA, in turn, we developed a precise fault-injection mechanism so we could emulate the entire SoC with specific faults present. For the gate level and below, we devised a mix of SPICE simulations, and for body bias effect evaluation we ran in-silicon evaluations at the laboratory of Professor Amano at Keio University.

This chapter is structured as follows. Since reliability threats and how such threats surface has been covered in the general introduction, Dynamically Reconfigurable Processors are briefly introduced. The next section directly dives into how the inherent architectural redundancy can be put to use to increase the reliability of computations, as well as how to test these techniques. In the following two sections, the focus then shifts to both ends of the abstraction layers by focussing on how to infer the device state at the transistor level and potentially also recovering from a faulty state using body biasing together with how decisions on the software or operating system level affect the transistor level. The last technical section before wrapping up then brings all levels together by highlighting the interplay between each layer and the synergistic gain thereby achieved.

2 Dynamically Reconfigurable Processors

Dynamically reconfigurable architectures, or short DRP, are a sub-category of so-called coarse-grained reconfigurable architectures (CGRA). Similar to Field Programmable Gate Arrays (FPGA), CGRAs are reconfigurable architectures; however, in contrast to FPGAs, CGRAs are reconfigurable on a far coarser level. That is, while FPGAs can efficiently map per-bit configurability, CGRAs only allow reconfigurability on word-sized units. While this restriction makes CGRAs unfavorable for random bit logic, CGRAs possess a far greater area and energy efficiency as the logic overhead for reconfigurability per bit is far lower. DRPs add the concept of dynamic reconfiguration to CGRAs by having on-chip memories for multiple configurations, or contexts, as instructions are often called in DRPs. As the keyword instruction already hints, DRPs resemble much more simple processors

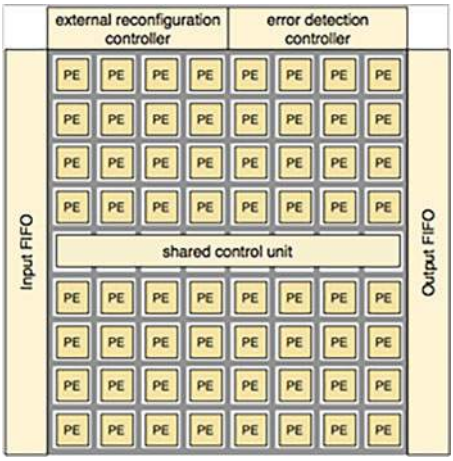


Fig. 2 Exemplary DRP instance with additional controllers and I/O buffers

than classical reconfigurable architectures, hence this reconfiguration mechanism is also often referred to processor-like reconfiguration.

DRPs at the point of writing date back more than 25 years which makes an exhaustive overview unfeasible. Instead, three different cited surveys shall give both a historical, functional, and up-to-date introduction to the field. De Sutter et al. [4] take a processor-centric view on CGRA architectures using the concept of instruction slots, that is logic where instructions can be executed. These units are connected using a simple form of interconnect like, e.g., nearest neighbor interconnect, and all have shared, or as De Sutter et al. describe them, distributed register files.

On the other hand, Hideharu Amano defines CGRAs and DRPs from a general hardware perspective. In [5], he defines a DRP to be an array of coarse-grained cells as depicted in Fig. 2, so-called PEs, consisting of one or multiple ALU and/or functional units (FU), a register file and a data manipulator [5]. The third and last survey cited for the purpose of an encompassing definition takes a similar approach as the authors of this chapter. In [6], Kiyoung Choi characterizes CGRA and by extension also DRPs via configuration granularity. All authors' definitions encompass an array of PEs and possess dynamic reconfiguration or processor-like execution and thus DRPs as architectural concept range from small reconfigurable DSP like blocks to many-core processors.

In theory, this allows the generalization of findings obtained in DRPs to be extended to far more complex brethren. In practice, however, the definition is restricted by precisely the architectural complexity as DRPs aim to be more energy efficient in more specialized fields other than, e.g., GPGPUs. This becomes also apparent in the general lack of complex caches and big register files, as well as simplistic, spatial interconnects that reduces both register file accesses and long and energy inefficient data transfers [4, 5]. For the purpose of this research project, this minimalism was a welcome attribute as it allowed an abstraction of far more complex architectures while maintaining generality. For this reason, we refer to the cited surveys [4–6] for comprehensive coverage of concrete DRP architectures.

3 Exploiting Architectural Redundancy for Increased Reliability

3.1 Realizing Low-Cost TMR Using PE Clusters

Among the most apparent aspects of DRPs is their regular structure. One of the first investigations published in [7] therefore sought to utilize the structural redundancy to increase DRPs' reliability by implementing the quasi-gold standard of fault-tolerance, triple modular redundancy (TMR). The biggest issue of TMR and also the reason why it is only used in critical systems is the prohibitively high cost, i.e., everything that is secured through TMR is triplicated. These triplicated copies then have to perform the exact same operation, and at given checkpoints or most commonly at the block level of the covered component, the outputs are compared. If an error surfaced, the correct result, as well as the faulty component, are determined through a majority vote. The big drawback of this technique is the high cost, both in circuit size since three copies are required, as well as in power consumption as all have to perform the same operation all the time. This makes TMR unviable for all but the most critical applications. With reconfigurable hardware, such as DRPs, however, hardware resource can be dynamically allocated. Given the addition of error detection components, the penalty of TMR can be severely reduced as resources do not have to be committed in a hard-wired fashion, but can be reassigned temporally, or, TMR could be dynamically used for specially flagged parts of a program only.

Figure 3 depicts a simplified representation of the Flexible Error Handling Module. It consists of an actual data error detection module, containing a three-input comparator. The comparator results are fed to the voter and the timing error detection. The voter determines the correct results through a majority vote and feeds the correct channel selection to the multiplexer which then forwards the result that

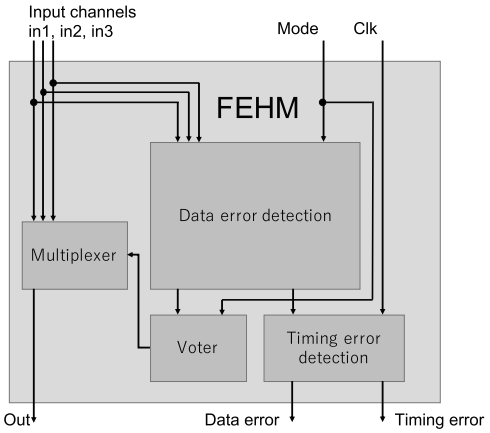


Fig. 3 The flexible error handling module (FEHM)

is now presumed to be correct to the next PE or out of the DRP. The timing error detection samples the comparison results in a double buffer on `Clk` the clock signal as well as on a slightly delayed clock signal. If the double buffer's contents on each sample are not the same, a timing error occurred and will be appropriately signaled. Similarly, if not all comparison results are equal in the first place, it will raise a data error signal. The entire module's functionality is controlled using the `Mode` signal. Using this signal, the FEHM can be turned off, to Duplicate with Comparison (DWC) mode or to full TMR mode.

This switch is central to the original goal of attaining TMR at lower cost: By making the mode signal part of the instruction word, not only does this free up TMR resources when TMR is not required, but it also allows for some degradation to DWC. Evaluations of this low-cost TMR evaluation showed that even if it is used in relatively primitive DRP architectures with very fine-grained data words, the additional hardware amounts for approximately a 6% increase in area. The power consumption, on the other hand, increased by about 7.5% which can be attributed to the constantly used XOR-OR trees and double buffers used for comparison and timing error detection.

3.2 DRPs as Redundancy for CPU Pipelines

CPUs as central control units in SoCs take a vital role and thus are of great interest for reliability. However, at the same time, they are among the most difficult components to harden against any type of fault if blunt and costly instruments such as TMR are avoided. The extreme degree of dynamism and control involved in CPUs make static redundancy schemes like TMR virtually mandatory if an error-free operation needs to be guaranteed. But if some tradeoffs are permissible, dynamic redundancy schemes can be alternatively used. Such tradeoffs can be for example an absolute time limit until recovery has to complete. In both cases, however, some form of spare component is required.

While DRPs will not be able to take over a CPU's main functions, they certainly could serve as spare compute pipeline [8], thus reducing the parts that need to be hardened using conventional methods. Placing a DRP into a processor's pipeline is not a novel idea such as [9] or [10] demonstrated and makes much sense from an acceleration point of view. However, as this chapter shall highlight, they might be a good pick concerning reliability as well. When used as a static redundancy as depicted in Fig. 4 (left), DRPs can make use of their structural redundancies to provide for additional samples computed in parallel to realize true TMR. The low-cost TMR method proposed in the previous section, on the other hand, can add an additional level of reliability so that the DRP's results can be trusted and false-positives effectively prevented. As dynamic redundancy or as a spare, the DRP can take over functionality if an error has been detected using other means as depicted in Fig. 4 (right). The viability of this approach has been validated in a model implementation inspired by ARM's Cortex-M3 microcontroller. This serves

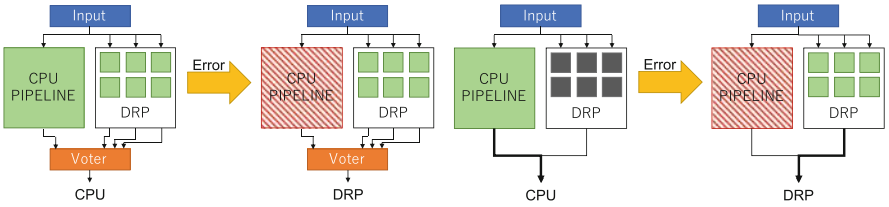


Fig. 4 DRP serving as static redundancy (left) and as dynamic redundancy (right)

as an interesting choice as ARM has its line of cores for safety-critical applications, the so-called ARM Cortex-R series with support for dual-core lock-step [11]. The results of this study as published in [8] showed that as long as support for division units is omitted in the DRP, the area overhead is far lower than the 100% overhead of an additional core, however, while of course leaving out other components to be secured separately. In this particular study, a 2 by 2 PE array, that is 4 PEs have been integrated into the CPU pipeline. Additionally, instructions and infrastructure to utilize the DRP have been added. Comparing the incurred overheads to a single-core implementation without any reliability measures, the area overhead for an implementation without hardware implemented division amounted to 20%. While this might not be an entirely fair comparison, division implementations in DRPs have a greater impact due to the far greater number of processing elements.

3.3 Dynamic Testing

In contrast to critical applications, SoCs often also accommodate non-essential functionality. For these applications, running all parts in TMR mode might be wasteful, yet a certain temporal assurance would be desirable. For example, in case of infotainment, brief dysfunction might be tolerable, but if functionality cannot be restored within a given amount of time, actual damage ensues. To avoid TMR or DWC for all applications and to implement time and probability based levels of reliability, we proposed a dynamic testing scheme for reconfigurable hardware.

Dynamic testing or also often called online testing as defined by Gao et al. [12] describes a testing method where for a known algorithm implemented in a certain component, input samples, and associated output samples are obtained and then recomputed separately. If the recomputation's results match the output samples, no error is present. If there is a mismatch, an error of the tested component is assumed.

Specifically using DRPs for dynamic testing has a big advantage: the choice between utilizing the temporal and spatial domains. Instead of competing with applications for resources on the DRP, dynamic testing resources can be allocated temporally and inserted interleaved with applications' instructions to be executed in a time-multiplexed fashion. By moving and interleaving into the time domain, testing becomes slower. However, for most non-critical applications, a couple of

seconds before a system returns to a functioning state can be tolerated. Furthermore, the spatial domain allows alternating the compute units used to recompute the samples, further making false-positives less likely apart from the error checking conducted during TMR usage.

While these two aspects make DRPs appealing for such testing schemes, time-multiplexing restricting testing to time-windows T_{TW} and further mapping into the temporal domain slowing down testing by a scaling factor s in combination with the probabilistic nature of error occurrence and detection make any estimation rather difficult. Therefore, Monte Carlo simulations can be used to estimate the behavior of dynamic testing accounting for all DRP specific aspects. For example, aspects such as reconfiguration overhead T_{OV} which has to be deducted from time-windows T_{TW} as well as scaling factors which reduces the number of samples that can be computed within one T_{TW} to detect a fault with an observation probability of q.

Consider Fig. 5, depicting a feasibility plot to detect a fault with an observation probability of $q = 10^{-5}$ and a reconfiguration overhead of 1 ms. The goal in this experiment was to detect such a fault within 2 s. The red striped regions indicate that here, it would take more than 2 s to detect the fault, whereas shades from white (fastest) to black indicate increasing detection latency DL. This result shows that even if the temporal domain is massively utilized at e.g. $s = 77$, the deadline of 2 s is still met at $DL = 1.7$ s with a time-window of 2 ms for computations thus allowing to use spatially extremely compact mappings for fault detection. This compaction also allows to further share the DRPs resources to conduct periodic checks of the

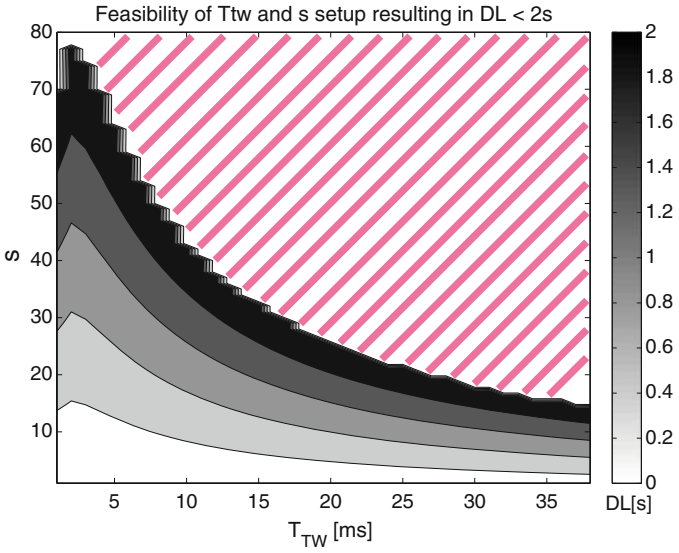


Fig. 5 Dynamic testing feasibility for a detection latency DL of 2 s by scaling factor s and time-window size for a fault with observation probability $q = 10^{-5}$, $T_{OV} = 1$ ms, a clock frequency of $F = 100$ MHz and 2σ confidence

entire surrounding SoC, expanding the reach of a reliable DRP to other system components as well.

3.4 Dynamic Remapping

Having various reliable ways to detect errors is vital as any reaction to a false-positive would just turn any reliability mechanism against itself. With low-cost TMR and dynamic testing, we have ways to detect errors and in the TMR case even to mask them. However, once a permanent fault is present and errors surface, TMR degrades to DWC, and dynamic testing is also limited to reasserting the error's presence over and over again. As DRPs are a class of reconfigurable hardware, to restore proper functionality, the applications have to be mapped anew avoiding faulty components. To do this, however, the remapping method and sufficient mapping resources are required.

In case of the FEHM equipped DRP used for our studies, two dimensions of redundancies can be utilized to run the application on unaffected PEs of the DRP. (1) spatially moving the application part of one faulty PE to a fault-free unused PE and (2) temporally adding the application part to an unaffected PE which is used for other application parts but still has the capacity to accommodate this part. As in DRPs the amount of instructions that can be stored and executed without external reconfiguration is limited, compensating for one or more faulty PEs can be a challenge in highly utilized scenarios. However, even if utilization is not critical, just moving parts around on the DRP will yield sub-optimal results, which is why the application mapping, that is resource allocation and scheduling needs to be rerun. This task, however, needs to be run on the SoCs CPU without obstructing normal operation.

To reduce the work-load of the SoC's CPU, we proposed an incremental remapping algorithm in [13]. First, the architecture graph is adjusted by removing the faulty components. Then, from this architecture graph, we extract a subgraph containing the affected PE and its vicinity. Similarly, the application graph is used to extract a subgraph containing only the application nodes mapped to the affected nodes in the architecture subgraph. With these two subgraphs, the mapping is then attempted as exemplarily depicted in Fig. 6. The mapping algorithm will try to first utilize the spatial dimension before resorting to the temporal dimension, i.e. prolonging execution time. If both dimensions do not have the resources to accommodate the application subgraph on the nodes of the pruned architecture subgraph, the architecture subgraph is enlarged by adding further neighboring nodes and remapping is retried until a new mapping has been found or the process fails altogether. If the process succeeds, the application now can run again without any errors occurring, even in non-TMR modes.

This prioritization of subgraph size over runtime, i.e., increasing subgraph size only if both dimensions cannot accommodate the application subgraph is arbitrary and other tradeoffs might be preferable. In this specific case, the priority was CPU

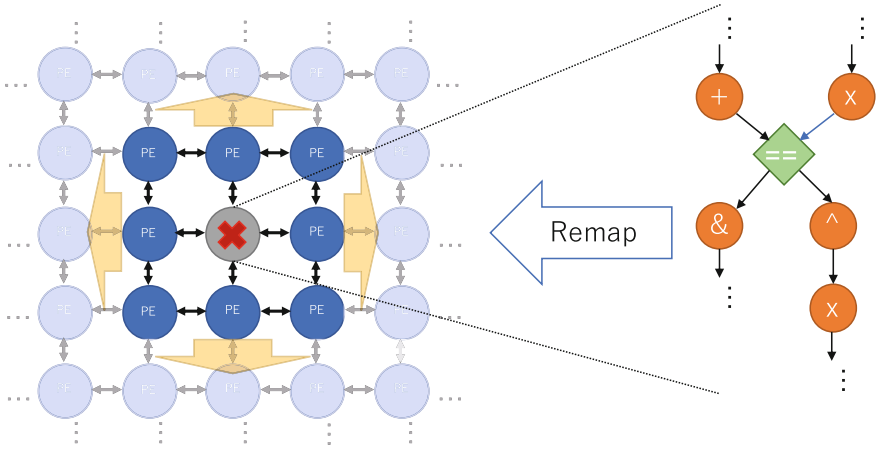


Fig. 6 Incremental remapping flow on architecture and application subgraphs to avoid faulty components. Starting on the direct neighborhood first, expanding if resources do not suffice

usage minimization, and therefore runtime and memory usage were prioritized by using the smallest subgraphs first at the expense of increased runtimes of the new mappings. For real-world applications, this needs to be carefully weighted as increased runtimes might not be viable.

3.5 Testing Reliability Schemes in Hardware

One of the big challenges of hardware manufacturing and particularly of implementing hardware-based countermeasures to reliability issues is testing and verification. Given the enormous number of input vectors and states, exhaustive testing via simulation is entirely unfeasible. While big commercial hardware emulators allow for a much greater design size and ease of use, they are also very costly. For small to medium-sized designs, FPGAs offer a sweet spot for prototype implementations. While simulations allow for easy fault injection but very slow simulation speeds, FPGAs offer speeds close to ASIC implementations but fault injection was virtually unfeasible.

To develop a prototyping platform, the Gaisler LEON3 SoC [14] served as a template into which the hardened DRP has been integrated. Parallel to this effort, different techniques for FPGA fault injection have been studied [15], culminating in the Static Mapping Library (StML) approach [16]. While instrumentation, i.e. RTL level insertion of faulty behavior allows unlimited choice in fault type and temporal behavior, it also requires for the RTL to be recompiled after each change. As the entire compilation and mapping process of our SoC took more than 4 h, this approach was abandoned. On the other hand, directly inserting faults into FPGA

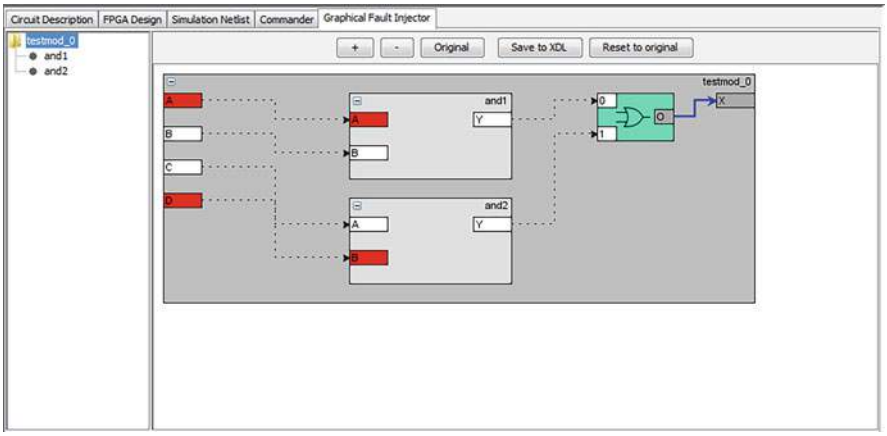


Fig. 7 StML GUI view of a sample logical AND component with fault injectable ports in each module

mappings or even the bitstream offers a simplistic way to create faulty versions of an FPGA mapping, but it offers no control over the type or location of the injected fault. This approach would not even guarantee that the FPGA mapping would behave in a faulty manner. Ideally, the exact fault location should be specifiable on RTL level to fully qualify the efficiency of the proposed architectural methods. To realize this, different intermediate results were utilized, primarily the FPGA's simulation netlist containing both RTL level structural information and FPGA mapping names in combination with the Xilinx Design Language (XDL) file containing the concrete FPGA mapping. By establishing a bidirectional link between the simulation netlist and the XDL file, StML enabled to pinpoint ports of module's implementation right down to the logic level to insert a stuck-at-zero or stuck-at-one fault. As the placed and routed XDL file can be directly altered, the only remaining step after fault injection is bitstream generation. A user-friendly GUI (Fig. 7) offering graphical representations of the implementation as well as a powerful command line interface allowed for both smooth experiment and extensive testing. Using this approach, we were able to reduce the fault-injection experiment time from hours to below 5 min, with most experiments done in below 2 min.

To showcase the viability of the proposed techniques, low-cost TMR, dynamic testing, dynamic remapping, and the FPGA prototype combined with the fault injection techniques have been successfully demonstrated at ICFPT in 2013 [17].

4 Device-Level State and Countermeasures

Below the architectural level, we studied opportunities to determine the state of semiconductor devices. Additionally, we also considered specific device-level

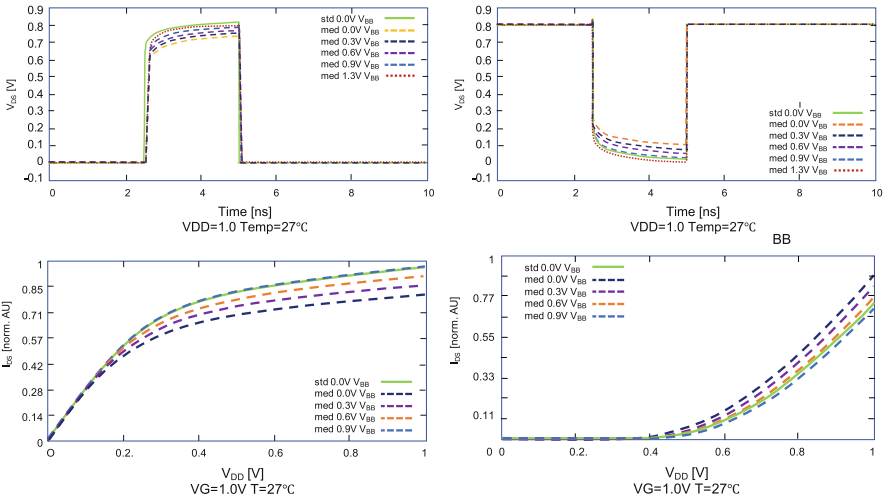


Fig. 8 Transient simulation for a single switching NMOS (upper left) and a single PMOS (upper right), as well as DC sweep of I_{DS} over $V_{DD} = V_{GS}$ for a NMOS (bottom left) as well as a PMOS transistor (bottom right), all using a medium degeneration model

countermeasures and their effects to put hardware into a more reliable state for tasks that require higher levels of reliability.

When considering how to obtain information on the state at the device level, a transistors' threshold voltage V_{TH} is a central variable [18] to consider. While of course, not all reliability phenomena manifest as an actual shift in V_{TH}, they can be modeled as such. For example, stuck-at faults are either a reduction to 0 V or ∞V of V_{TH} or even changes in the drive current and subsequent timing faults can be viewed as such. With the semiconductor world moving either towards FinFET or FDSOI technologies, we investigated the options of FDSOI processes such as ST Microelectronics Ultra Thin Body and Box Fully Depleted Semiconductor on Insulator (UTBB-FDSOI) technology [19]. While being a planar technology, it is manufactured in a triple well process, shielding the transistor body against the substrate using a diode in reverse direction. The transistor is manufactured using a fully depleted channel which allows for further scaling to compete with FinFET processes. One of the main advantages of FDSOI technologies is that the insulated transistor body allows for very high biasing voltages previously unfeasible as it would have shorted the transistor to the substrate. As this thin body with the thin box construction equipped with a separate body electrode acts as a second gate, it is ideal to adjust V_{TH} dynamically after manufacturing. The adjustment of a transistor via this second gate is also called body biasing.

To study the possibilities to use body biasing to detect faults or even faults building up, SPICE level models have been considered. Figure 8 depicts the transient and DC analysis of a medium degeneration transistor-level model. The left side in Fig. 8 depicts an NMOS transistor whereas the right side depicts a PMOS

transistor. In each graph, there are several plots: std $0.0VV_{BB}$, that is a perfectly functioning transistor without any body bias applied, med $0.0VV_{BB}$ transistor with a medium V_{TH} increase and no body biasing, and then several variants of the defective transistor with increasing levels of body biasing. When comparing std $0.0VV_{BB}$ to med $0.0VV_{BB}$, it immediately becomes apparent that there is a significant gap in the rate at which the signal rises (top two graphs) and signal level, as well as a strong difference in drive current (bottom two graphs). The effects of a V_{TH} (about 45 mV NMOS and 40 mV PMOS) shift of this magnitude are, of course, relative to the operating conditions. If e.g. a couple of such transistors would be used somewhere on a critical path within a high-performance circuit, it would surely fail. On the other hand, if the circuit is used far from timing limits or if only a single transistor is considered, the effect might be barely noticeable. Given an on-chip test circuit or a known critical path, they can be used in conjunction with body biasing to measure degenerative effects. To perform such post-manufacturing bias, ideally each chip should be tested after production with a sweep over body bias levels as described in [20], with the minimum body bias, that is the maximum reverse body bias (the circuit's timing is intentionally slowed down), at which the circuit checked out functional written to a non-volatile memory. Later on, this minimum bias point can be used as a reference, i.e., if the chip or the tested component needs a higher level of body bias corrected for temperature, then some degeneration occurred. If the circuit is designed with reasonable margins, a build-up until an actual fault occurs can be thereby detected.

Similarly, body bias also allows pushing the circuit back conforming to specification. The effect depicted in Fig. 8 would be catastrophic for any performance-oriented component. However, this medium degeneration case has been chosen specifically so that corrective measures can be taken without special electrical precautions, which is up to a V_{BB} of 1.3 V in most processes. However, it should be noted that this also leads to significantly increased leakage levels and would be unfeasible for an entire chip. This being said, it neatly complements DRPs' architectural granularity, i.e. one PE would be coarse enough to mitigate the overheads of an individual body bias domain, yet it is small enough to keep the leakage overhead of strong forward biases down [21]. Additionally, finer steps of, e.g., 100 mV should be used to detect shifts in V_{TH} early on.

5 Synergistic Effects of Cross-Layer Approaches

The question following from the previous section is whether to use architectural approaches or device-level countermeasures to achieve a certain reliability objective is an extremely complex and multivariate problem. Beyond the question whether or not to use a specific technique, there are additional variables such as time, i.e. when to use these techniques, extend, that is in what parts to use them and also in regard to criticality, what techniques and with which parameters could be used at all and to what end?

In a very insightful collaboration with the FEHLER project (chapter "Soft Error Handling for Embedded Systems using Compiler-OS Interaction"), their static analysis of program criticality provided powerful means to determine key portions for reliable execution at the application level [22]. By annotating source code with keywords indicating the respective criticality, only those parts marked as critical will be additionally secured using reliability techniques. It thus was not only a great fit for selective low-cost TMR on the hardened DRP, but beyond that offered a proof-of-concept of mixed-criticality applications along with the means to identify portions critical for reliability. In [22], the targeted application was an h264 decoder. As an entertainment application, the primary metric is whether the service is provided at a certain perceived quality level above which actually occurring errors are irrelevant as they are imperceivable.

On the other end of the scale, device state monitoring allows to assess the physical state of a SoC and also its progression over time. On the architectural level, low-cost TMR or DWC allows for continuous checking, whereas dynamic testing makes sure that errors are not left undetected indefinitely, both providing vital information to potential agents. However, as shall be explored below, reactive measures cannot be determined on one layer alone.

Once the device-level state is known, this information can be used on every abstraction layer above. If for example degradation has been detected, this information can be used to minimize physical stresses by using a combination of supply voltage V_{DD} and body bias V_{BB} [2]. As proposed in the previous subsection, a concrete proposal is to counter V_{TH} drift, that is usually V_{TH} becoming larger, by using a forward body bias. As, however, Federspiel et al. found in [2], this will also increase effects like Hot Carrier Injection (HCI) stress which in turn can cause a decrease in drive current. This could lead to a feedback loop as with the method described in Sect. 4, this would appear like a V_{TH} increase and cause more forward bias to be applied, further increasing HCI stress. Thus, such action needs to be a concerted effort on the operating system level with a full view of the system state and the resources available.

For this reason, countermeasures could encompass several different options from the set of available countermeasures with the primary distinction on lifetime extension or securing error-free functionality in the presence of faults. In both cases, it should be noted that both distinctions are only two different takes on graceful degradation. In case the primary goal of reactive measures is lifetime extension, measures which incur less physical stress should be taken. If e.g. the application allows for some degeneration of the service level such as the aforementioned h264 decoding, less effort can be spent on uncritical parts of an application or it could be mapped alongside another application on a DRP. If error-free functionality is the primary goal because, e.g., the application is critical and does not allow for any degeneration, there is a two-step cascade. If the application can be remapped to fault-free components, this should be prioritized. If the resources do not permit remapping or if no resources are left, the SoC can attempt to mitigate the fault through, e.g., a forward body bias at the expense of a potentially shortened lifetime.

In all cases, however, it is clear that information from the application layer, the operating system, i.e. knowledge about what else is running on the SoC, the

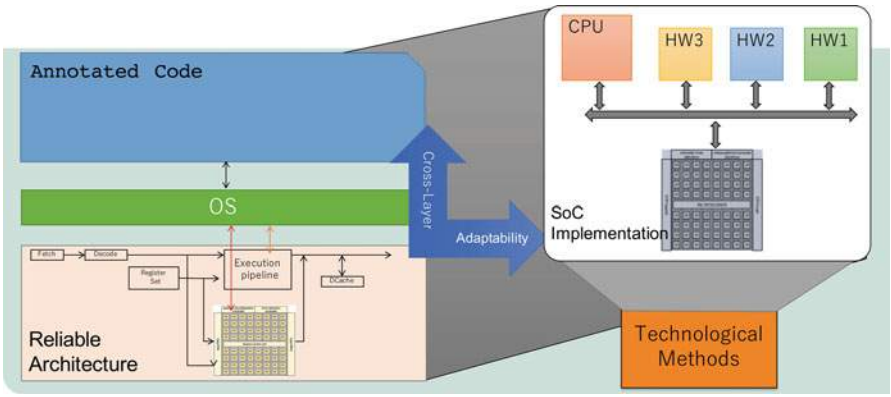


Fig. 9 Using information of all abstraction layers to realize more reliable and efficient SoC

architectural layer, what resources are available, which resources are inoperable, etc., as well as the device level, are all key to determine the optimal response. In the specific example visualized in Fig. 9, we start at the application layer by assuming reliability annotated source code. Using this source code, an appropriate mapping for example with low-cost TMR onto the DRP can be determined. Additionally, the OS might then go ahead to issue its execution without any special circuit-level tuning, i.e. increasing supply voltage or forward bias to add timing margins. Similarly, a mapping onto the CPU pipeline could be more suitable where the OS then might opt for extra forward bias as some degradation has been previously detected and the application is realizing important functionality. Not only does such a cross-layer approach as visualized in Fig. 9 help to achieve the reliability objectives, but it also is capable of more than what can be achieved on one layer at a time [23]. In this concrete example, the incorporation of multiple layers and multiple methods at specific layers allows to tailor reliability measures to requirements. Device-level information enables the system to act proactively as many phenomena can be detected at this layer in the build-up phase. Once the device layer degenerates, actors such as body biasing allow a system to restore or prolong functionality in the presence of faults.

6 Conclusion

Over a generous 6-year period in which this project was funded, the possibilities to use DRPs for increased reliability were extensively studied and also tested in prototype implementations at a functional level. This research revealed that DRPs are not only well suited for tasks that require TMR like reliability, but they can be used in numerous ways to improve the reliability of entire SoCs as well. Their simple and efficient structure allowed to research new and efficient concepts such as

dynamic remapping or body biasing for device-level sensing and countermeasures. While DRPs are still undeservingly viewed as a kind of fringe architecture concept, most of the insights gained through such architectures are easily transferable to multi- or many-core SoCs. This project showed that far more can be done in regard to reliability if multiple abstraction layers are considered in a cross-layer approach. While common wisdom still is to use TMR whenever software people use terms such as error-free or fault-tolerant, this project showed multiple options how to incorporate more specific application requirements and how to translate this into adequate reliability measures. Or in simpler terms, just-safe-enough responses to the reliability threats.

Acknowledgments As the funding agency of this project, the authors express their gratitude to the DFG for the generous funding throughout the project duration under RO-1030/13 within the Priority Program 1500. The authors also would like to thank STMicroelectronics for the cooperation on FDSOI technology. In regard to the exploration of technological aspects, the authors would also like to express their deep gratitude to the lab of Prof. Hideharu Amano at Keio University and its partnering institutions. It was a tremendous help to see to possibilities of FDSOI in silicon very early on. Finally, the authors would like to thank all the collaborating projects, persons, and especially all the alumni that were involved in this project for the fruitful discussions and interesting exchanges.

References

1. Herkersdorf, A., Aliee, H., Engel, M., Glaß, M., Gimmler-Dumont, C., Henkel, J., Kleeberger, V.B., Kochte, M.A., Kühn, J.M., Mueller-Gritschneder, D., et al.: Resilience articulation point (RAP): cross-layer dependability modeling for nanometer system-on-chip resilience. Microelectron. Reliab. **54**(6–7), 1066–1074 (2014)
2. Federspiel, X., Angot, D., Rafik, M., Cacho, F., Bajolet, A., Planes, N., Roy, D., Haond, M., Arnaud, F.: 28 nm node bulk vs FDSOI reliability comparison. In: 2012 IEEE International Reliability Physics Symposium (IRPS) pp. 3B–1. IEEE, Piscataway (2012)
3. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., et al.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(1), 8–23 (2012)
4. De Sutter, B., Raghavan, P., Lambrechts, A.: Coarse-grained reconfigurable array architectures. In: Handbook of Signal Processing Systems, pp. 427–472. Springer, Berlin (2019)
5. Amano, H.: A survey on dynamically reconfigurable processors. IEICE Trans. Commun. **89**(12), 3179–3187 (2006)
6. Choi, K.: Coarse-grained reconfigurable array: Architecture and application mapping. IPSJ Trans. Syst. LSI Des. Methodol. **4**, 31–46 (2011)
7. Schweizer, T., Schlicker, P., Eisenhardt, S., Kuhn, T., Rosenstiel, W.: Low-cost TMR for fault-tolerance on coarse-grained reconfigurable architectures. 2011 International Conference on Reconfigurable Computing and FPGAs, pp. 135–140. IEEE, Piscataway (2011)
8. Lübeck, K., Morgenstern, D., Schweizer, T., Peterson, D., Rosenstiel, W., Bringmann, O.: Neues Konzept zur Steigerung der Zuverlässigkeit einer ARM-basierten Prozessorarchitektur unter Verwendung eines CGRAs. In: MBMV, pp. 46–58 (2016)
9. Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R.: ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: International Conference on Field Programmable Logic and Applications, pp. 61–70. Springer, Berlin (2003)

10. Hoy, C.H., Govindarajuz, V., Nowatzki, T., Nagaraju, R., Marzec, Z., Agarwal, P., Frericks, C., Cofell, R., Sankaralingam, K.: Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 203–214. IEEE, Piscataway (2015)
11. Arm Cortex-R Series Processors. https://developer.arm.com/products/processors/cortex-r. Accessed 8 March 2019
12. Gao, M., Chang, H.M., Lisherness, P., Cheng, K.T.: Time-multiplexed online checking. IEEE Trans. Comput. **60**(9), 1300–1312 (2011)
13. Eisenhardt, S., Küster, A., Schweizer, T., Kuhn, T., Rosenstiel, W.: Spatial and temporal data path remapping for fault-tolerant coarse-grained reconfigurable architectures. In: 2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, pp. 382–388. IEEE, Piscataway (2011)
14. LEON3 Processor. https://www.gaisler.com/index.php/products/processors/leon3?task=view&id=13
15. Schweizer, T., Peterson, D., Kühn, J.M., Kuhn, T., Rosenstiel, W.: A fast and accurate fpga-based fault injection system. In: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 236–236. IEEE, Piscataway (2013)
16. Peterson, D., Bringmann, O., Schweizer, T., Rosenstiel, W.: StML: bridging the gap between FPGA design and HDL circuit description. In: 2013 International Conference on Field-Programmable Technology (FPT), pp. 278–285. IEEE, Piscataway (2013)
17. 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, Piscataway (2013)
18. Maricau, E., Gielen, G.: CMOS reliability overview. In: Analog IC Reliability in Nanometer CMOS, pp. 15–35. Springer, Berlin (2013)
19. Pelloux-Prayer, B., Blagojević, M., Thomas, O., Amara, A., Vladimirescu, A., Nikolić, B., Cesana, G., Flatresse, P.: Planar fully depleted SOI technology: The convergence of high performance and low power towards multimedia mobile applications. In: 2012 IEEE Faible Tension Faible Consommation, pp. 1–4. IEEE, Piscataway (2012)
20. Okuhara, H., Ahmed, A.B., Kühn, J.M., Amano, H.: Asymmetric body bias control with low-power FD-SOI technologies: modeling and power optimization. IEEE Trans. Very Large Scale Integr. Syst. **26**(7), 1254–1267 (2018)
21. Kühn, J.M., Ahmed, A.B., Okuhara, H., Amano, H., Bringmann, O., Rosenstiel, W.: MuCCRA4-BB: a fine-grained body biasing capable DRP. In: 2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX), pp. 1–3. IEEE, Piscataway (2016)
22. 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). http://doi.acm.org/10.1145/2536747.2536753
23. Herkersdorf, A., Engel, M., Glaß, M., Henkel, J., Kleeberger, V.B., Kochte, M., Kühn, J.M., Nassif, S.R., Rauchfuss, H., Rosenstiel, W., et al.: Cross-layer dependability modeling and abstraction in system on chip. In: Workshop on Silicon Errors in Logic-System Effects (SELSE) (2013)

Dependable Software Generation and Execution on Embedded Systems

Florian Kriebel, Kuan-Hsun Chen, Semeen Rehman, Jörg Henkel, Jian-Jia Chen, and Muhammad Shafique

1 Overview

An overview of the chapter structure and the connection of the different sections is illustrated in Fig. 1. Soft error mitigation techniques like [17, 29] have shown that the software layer can be employed for enhancing the dependability of computing systems. However, to effectively use them, their overhead (e.g., in terms of power and performance) has to be considered. This also includes the option of adapting to different output accuracy requirements and inherent resilience against faults of different applications, for which appropriate metrics considering information from multiple system layers are required. Therefore, we start with a short overview of reliability and resilience modeling and estimation approaches, which not only focus on the functional correctness (like application reliability and resilience) but also consider the timeliness, i.e., determining the change of the timing behavior according to the run-time dependability, and providing various timing guarantees for real-time systems. They are used to evaluate the results of different dependable

F. Kriebel (✉) · M. Shafique
Technische Universität Wien (TU Wien), ECS (E191-02), Institute of Computer Engineering, Wien, Austria
e-mail: florian.kriebel@tuwien.ac.at; muhammad.shafique@tuwien.ac.at

K.-H. Chen · J.-J. Chen
Technische Universität Dortmund, Lehrstuhl Informatik 12, Dortmund, Germany
e-mail: kuan-hsun.chen@tu-dortmund.de; jian-jia.chen@cs.uni-dortmund.de

S. Rehman
Technische Universität Wien (TU Wien), ICT (E384), Wien, Austria
e-mail: semeen.rehman@tuwien.ac.at

J. Henkel
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: henkel@kit.edu

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_6

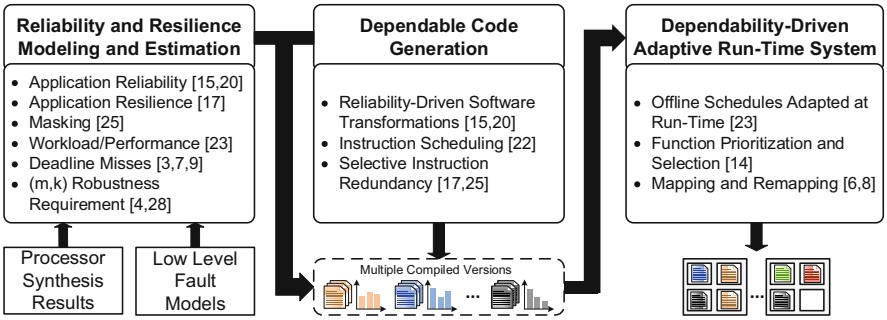


Fig. 1 Overview

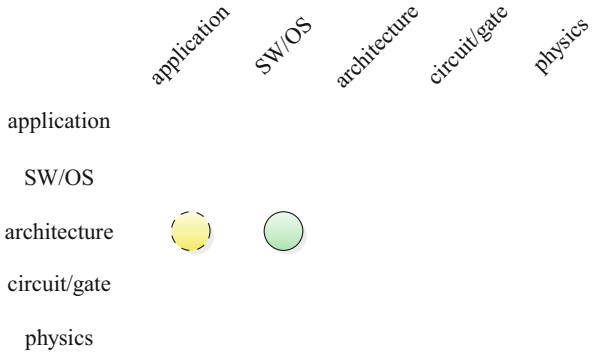


Fig. 2 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

code generation approaches, like *dependability-driven software transformations* and *selective instruction redundancy*. This enables generation of multiple compiled code versions of an application realizing different performance/energy vs. dependability trade-offs. The evaluation results and the different versions are then used by a *dependability-driven adaptive run-time system*. It considers *offline* and *online* optimizations, for instance, for selecting appropriate application versions and adapting to different workloads and conditions at run-time (like fault rate, aging, and process variation). Thereby, it finally enables a dependable execution of the applications on the target system.

As, however, not all systems are general-purpose, towards the end of the chapter an example design of a video processing system is included, which illustrates different approaches for application-specific dependability.

Embedding this chapter's content in the scope of this book and the overall projects [12, 14], the main contributions lie on the application, SW/OS, and architectural layers as illustrated in Fig. 2.

2 Dependability Modeling and Estimation

Modeling dependability at the software layer is a complex task as parameters and effects of different systems layers have to be taken into account. For an accurate yet fast evaluation of application dependability, information from lower system layers has to be considered, while abstracting it in a reasonable way to also allow for a fast estimation. For this purpose, different aspects have been separated into distinct metrics focusing on individual phenomena, as discussed below.

- The **Instruction Vulnerability Index (IVI)** [19, 25] focuses on the error probability of each instruction when being executed on different components/pipeline stages of a processor by analyzing their *spatial and temporal vulnerabilities*. This requires an analysis of *vulnerable bits* as well as *vulnerable time period*, i.e., the residence times of instructions in different components, while considering micro-architecture dependent information from the lower layers like the *area consumption* of different components and the *probability* that an error is observed at their output (see Fig. 1). The IVI of individual instructions can then be combined to estimate the vulnerability at higher granularity (e.g., Function Vulnerability Index—FVI). In this case, the susceptibility towards application failures and incorrect application outputs can be considered as well, for instance by classifying instructions into *critical* and *non-critical* ones, which is important if deviations in the application output can be tolerated.
- As not all errors occurring during the execution of an application become visible to the user due to *data flow and control flow masking*, the **Instruction Error Masking Index (IMI)** [31] provides probabilistic estimates whether the erroneous output of an instruction will be masked until the visible output of an application.
- The **Instruction Error Propagation Index (EPI)** [31] captures the effects of errors not being masked from the time of their generation until the final output of an application. It analyzes the propagation effects at instruction granularity and quantifies the impacts of the error propagation and how much it affects the final output of an application.
- Based on the information theory principles, the **Function Resilience** model [24] provides a probabilistic measure of the function's correctness (i.e., its output quality) in the presence of faults. In contrast to the *IVI/FVI*, it avoids exposing the application details by adopting a black-box modeling technique.
- The **Reliability-Timing Penalty (RTP)** [23] model jointly accounts for the *functional correctness* (i.e., generating the correct output) and the *timing correctness* (i.e., timely delivery of an output). In this work, we studied RTP as a linear combination of functional reliability and timing reliability, where the focus (functional or timing correctness) can be adjusted. However, it can also be devised through a non-linear model depending upon the design requirements of the target system.
- The **(m,k) robustness constraint** model [4, 35] quantifies the potential inherent safety margins of control tasks. In this work, several error-handling approaches

guarantee the minimal frequency of correctness over a static number of instances while satisfying the hard real-time constraints in the worst-case scenario.
- The **Deadline-Miss Probability** [3, 9, 34] provides a statistical argument for the probabilistic timing guarantees in soft real-time systems by assuming that after a deadline miss the system either discards the job missing its deadline or reboots itself. It is used to derive the **Deadline-Miss Rate** [7], which captures the frequency of deadline misses by considering the backlog of overrun tasks without the previous assumption of discarding jobs or rebooting the system.

A more detailed description of the different models as well as their corresponding system layers are presented in chapter "Reliable CPS Design for Unreliable Hardware Platforms".

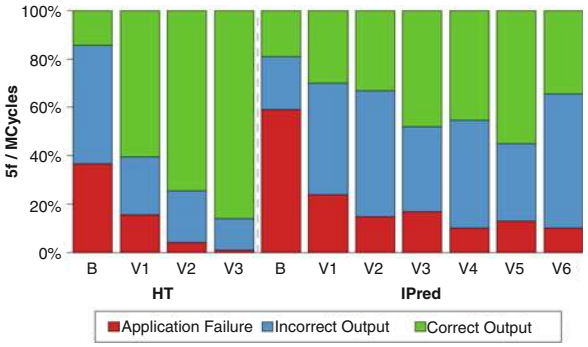
3 Dependability-Driven Compilation

Considering the models and therewith the main parameters affecting the dependability of a system, several mitigation techniques are developed, which target to improve the system dependability on the software layer. Three different approaches are discussed in the following.

3.1 Dependability-Driven Software Transformations

Software transformations like loop unrolling have mainly been motivated by and analyzed from the perspective of improving performance. Similarly, techniques for improving dependability at the software level have mainly focused on error detection and mitigation, e.g., by using redundant instruction executions. Therefore, the following dependability-driven compiler-based software transformations [19, 25] can be used to generate different application versions, which are identical in terms of their functionality but which provide different dependability-performance trade-offs.

- *Dependability-Driven Data Type Optimization:* The idea is to implement the same functionality with different data types, targeting to reduce the number of memory load/store instructions (which are critical instructions due to their potential of causing application failures) and their predecessor instructions in the execution path. However, additional extraction/merging instructions for the data type optimization have to be taken care of when applying this transformation.
- *Dependability-Driven Loop Unrolling:* The goal is to find an unrolling factor (i.e., loop body replications), which minimizes the number of critical instructions/data (e.g., loop counters, branch instructions) that can lead to a significant deviation in the control flow causing application failures. This reduction, however, needs to be balanced, e.g., with the increase in the code size.



Appl.	Ver.	Description
	B	Baseline
HT	V1	B+Common Expression Elimination
	V2	V1+Loop Unrolling (Factor 2)
	V3	V1+Complete Loop Unrolling (Factor 8)
IPred	V1	B+Loop Unrolling (Factor 4)
	V2	Data type optimization (32bit loads instead of 8bit)
	V3	B+Loop Unrolling (Factor 16+4)
	V4	V1+Loop Unrolling (Factor 16)
	V5	V2+calculation reordering
	V6	V3+calculation reordering

Fig. 3 Fault injection results for two applications and the generated application versions (adapted from [25])

- *Reliability-Driven Common Expression Elimination and Operation Merging:* The idea of eliminating common expressions is to achieve performance improvement due to less instructions being executed and therefore less faults being able to affect an application execution. However, excessively applying this transformation might lead to register spilling or longer residence times of data in the registers. Therefore, it needs to be evaluated carefully whether eliminating a common expression leads to a vulnerability reduction or whether the redundancy implied by a re-execution provides a benefit.
- *Reliability-Driven Online Table Value Computation:* The goal of the online table value computation is to avoid long residence times of pre-computed tables in the memory, where the values can be affected by faults and can therefore affect a large set of computations. This needs to be traded off against the performance overhead (and therefore increased temporal vulnerability) of online value computation.

As the transformations listed above also imply certain side effects (e.g., increased code size, additional instructions), they need to be applied carefully. We evaluate the above techniques using an *instruction set simulator-based fault injection approach*, where faults can be injected in different processor components (e.g., register file, PC, ALU, etc.) considering their area. It supports injecting a single or multiple faults per experiment, where each fault can itself corrupt a single or multiple bits. The results for two example applications from the *MiBench* benchmark suite [13] are shown in Fig. 3. They illustrate the effectiveness of the proposed transformations, e.g., for the "HT" application by the reduction of the application failures and incorrect outputs generated when comparing the *Baseline* application version and *V3*.

Finally, the dependability-driven software transformations are not only useful as a standalone technique, but can also be combined with other error mitigation techniques. For example, by reducing the number of instructions accessing the

memory, they can help reduce the required checking instructions in [29], and thereby lead to a performance improvement.

3.2 Dependability-Driven Instruction Scheduling

Instruction scheduling can significantly affect the temporal vulnerability of instructions and data, as it determines their residence time in different processor components. To improve the dependability of an application, several problems have to be addressed, which usually do not have to be considered for a performance-oriented instruction scheduling:

1. *Critical instructions should not be scheduled after multi-cycle instructions* or instructions potentially stalling the pipeline as this increases their temporal vulnerability;
2. *High residence time* (and therefore temporal vulnerability) of data in registers/memory;
3. *High spatial vulnerability*, e.g., as a consequence of using more registers in parallel.

Therefore, the dependability-driven instruction scheduling in [21, 22] estimates the vulnerabilities, and separates the instructions into *critical* and *non-critical* ones statically at compile-time before performing the instruction scheduling. Afterwards, it targets minimizing the application dependability by minimizing the spatial and temporal vulnerabilities while avoiding scheduling critical instructions after multi-cycle instructions to reduce their residence time in the pipeline. These parameters are combined to an evaluation metric called *instruction reliability weight*, which is employed by a *lookahead-based heuristic* for scheduling the instructions. The scheduler operates at the basic block level and considers the reliability weight of an instruction in conjunction with its dependent instructions to make a scheduling decision. In order to satisfy a given performance overhead constraint, the scheduler also considers the performance loss compared to a performance-oriented instruction scheduling.

3.3 Dependability-Driven Selective Instruction Redundancy

While the dependability-driven software transformations and instruction scheduling focus on reducing the vulnerability and critical instruction executions, certain important instructions might still have to be protected in applications being highly susceptible to faults. Therefore, it is beneficial to *selectively protect important instructions using error detection and recovery techniques* [24, 31], while saving the performance/power overhead of protecting every instruction.

To find the most important instructions, the error masking and error propagation properties as well as the instruction vulnerabilities have to be estimated. These results are used afterwards to prioritize the instructions to be protected, considering the performance overhead and the reliability improvement. For this, a *reliability profit function* is used, which jointly considers the protection overhead, error propagation and masking properties and the instruction vulnerabilities. The results of this analysis are finally used to select individual or a group of instructions, which maximize the total reliability profit considering a user-provided tolerable performance overhead.

4 Dependability-Driven System Software

Based on the dependability modeling and estimation approaches and the dependability-driven compilation techniques, *multiple code versions* are generated. These code versions exhibit distinct performance and dependability properties while providing the same functionality. They are then used by the **run-time system** for exploring different *reliability-performance trade-offs* by selecting appropriate application versions while adapting to changing run-time scenarios (e.g., different fault rates and workloads) for single- and multi-core systems.

4.1 Joint Consideration of Functional and Timing Dependability

The key requirement of many systems is producing correct results, where a (limited) time-wise overhead is oftentimes acceptable. However, for real-time (embedded) systems both the *functional dependability* (i.e., providing correct outputs even in the presence of hardware-level faults) and the *timing dependability* (i.e., providing the correct output before the deadline) play a central role and need to be considered jointly trading-off one against the other [23, 27]. To enable this, multiple system layers (i.e., compiler, offline system software, and run-time system software) need to be leveraged in a *cross-layer framework* to find the most effective solution [15]. For an application with multiple functions, the problem is to compose and execute it in way that jointly optimizes the functional and timing correctness. For this, the *RTP* (see Sect. 2) is used as an evaluation metric.

Figure 4a presents an overview of our approach. It is based on multiple function versions generated by employing the approaches described in Sect. 3, where additionally even different algorithms might be considered. As an example, a *sorting* application is illustrated in Fig. 4b, where the vulnerability of different algorithms and implementations as well as their execution times are compared showing different trade-offs. For generating the versions, a dependability-driven

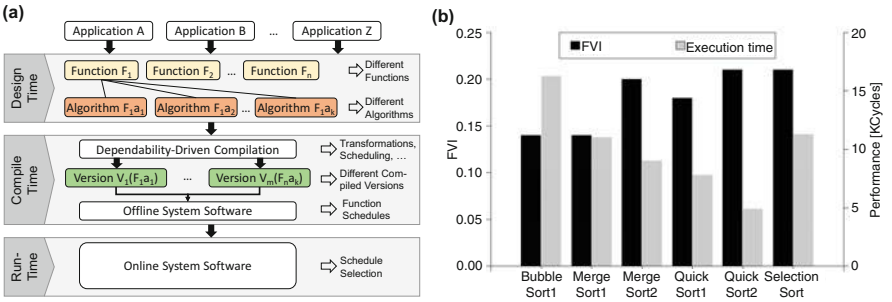


Fig. 4 (**a**) Overview of the design-time, compile-time, and run-time steps for generating different function/application versions. (**b**) Different algorithms and implementations for sorting (adapted from [23])

compilation process is used, given different implementations and a tolerable performance overhead to limit the design space. Then, only a limited number of versions from the *pareto-frontier* are selected, representing a wide spectrum of solutions.

In the next step, a *Dependability-Driven Offline System Software* generates **schedule tables** by minimizing the expected RTP. For the execution time, a probability distribution is considered, since it is not constant for all functions. For applications with only one function, the version minimizing the RTP (based on a weighting parameter) can be found by analyzing its probability for deadline misses and its reliability. For applications with multiple functions, it is required to consider that the selected version of a function is dependent on the functions executed earlier, e.g., if they finish early, a high-reliability version with a longer execution time can be selected. Therefore, a *dynamic version selection scheme* is adopted, where schedule tables are prepared offline and the scheduler selects appropriate function versions depending on the run-time behavior. Selecting a version for a particular function depends on both the functions executed *earlier*, and the functions executed *afterwards* (i.e. the predecessor and the successor functions in the execution path). The schedule tables are filled from the last function to be executed and remaining entries are added successively later, where the properties of earlier functions have to be explored and later functions can be captured by a lookup in the already filled parts of the table.

At run-time, a *Dependability-Driven Run-time System Software* selects an appropriate function version from the schedule table depending on the RTP. To execute the corresponding function, dynamic linking can be used. At the start of an application, the RTP is zero and the remaining time is the complete time until the deadline, as no function has been executed so far. With these parameters, the entry is looked up in the schedule table and the corresponding function version is executed. When one of the following functions need to be executed, the RTP observed so far is accumulated and the remaining time until the deadline is calculated. Afterwards, the corresponding table lookup is performed and a version is selected. To ensure

the correctness of the schedule tables, they should be placed in a protected memory part. As they, however, might become large, the size of the table can be reduced by removing redundant entries and entries where the RTP difference is too small. However, in this chapter, we assume that the system software is protected (for instance, using the approaches described in the OS-oriented chapters) and does not experience any failures.

In case the ordering of function executions is (partially) flexible, i.e., no/only partial precedence constraints exist, this approach can be extended by a function prioritization technique [27].

4.2 Adaptive Dependability Tuning in Multi-Core Systems

While Sect. 4.1 mainly focused on single-core systems and transient faults, the following technique will extend the scope towards multi-core systems and reliability threats having a permanent impact on the system (like process variation and aging). Thereby, different workloads on the individual cores might further aggravate the imbalance in core frequencies, which already preexists due to process variation. Consequently, *a joint consideration of soft errors, aging, and process variation is required* to optimize the dependability of the system. The goal is to achieve resource-efficient dependable application execution in multi-core systems under core-to-core frequency variation.

In a multi-core system, the software layer-based approaches can be complemented by *Redundant Multithreading (RMT)*, which is a hardware-based technique that executes redundant threads on different cores. An application can be executed with either Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR). This broadens the mitigation solutions against the above-mentioned dependability threats, but also demands for the following problems to be solved [26].

1. The activation/deactivation of RMT has to be decided based on the properties (i.e., vulnerability, masking, performance) of the concurrently executing applications, the allowed performance overhead, and the error rate.
2. Mapping of (potentially redundant) threads to cores at run-time needs to consider the cores' states.
3. A reliable code version needs to be selected based on the performance variations of the underlying hardware and the application dependability requirements.

These problems are addressed by employing two key components: (1) a *Hybrid RMT-Tuning* technique, and (2) a *Dependability-Aware Application Version Tuning and Core Assignment* technique.

The **Hybrid RMT-Tuning** technique considers the performance requirements and vulnerability of the upcoming applications in combination with the available cores and history of encountered errors. It estimates the RTP of all applications, activates RMT for the one with the highest RTP in order to maintain the history, and

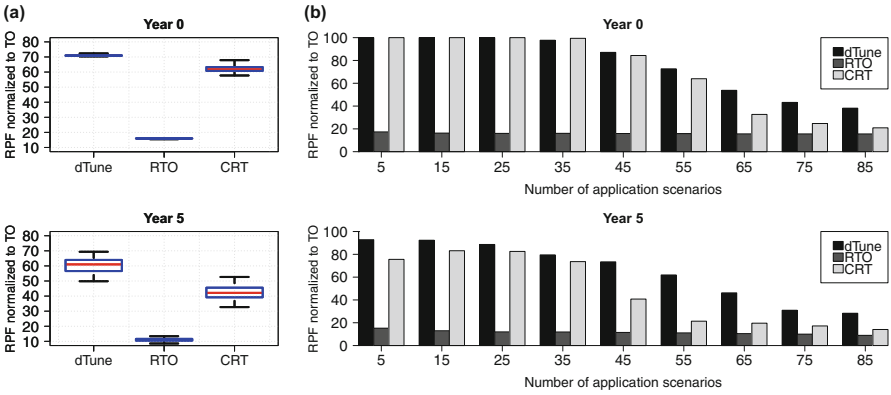


Fig. 5 (**a**) RPF improvements of *dTune*, *RTO*, and *CRT* normalized to *TO* for different aging years summarizing different chips and workloads. (**b**) RPF improvements detailing different workloads (adapted from [28])

takes RMT activation decisions based on the available cores and recent error history. For applications with RMT activated, the fastest compiled code version is selected.

After the RMT mode is decided for each application, the **Dependability-Aware Application Version Tuning and Core Assignment** is performed. It starts with an initial decision on the application version for applications where RMT has not been activated, considering their vulnerability and deadline. Then, the core allocation/mapping is performed, which takes the performance variations of individual cores (caused by process variation and aging) into account. It starts with the applications having the highest RTP and intends to allocate cores with similar performance properties to all redundant copies while also considering their distance. Finally, the application versions selected in the earlier step are tuned to improve the RTP further. Since the allocated core is now known, the potential for improving the dependability is evaluated considering the application's deadline.

Figure 5 shows the results of this approach (**dTune**) for different number of applications and different years. For the evaluation, a multi-core system with 10×10 ISA-compatible homogeneous RISC cores is used. These cores differ in their performance characteristics due to aging, where we consider NBTI-induced aging [1], and process variation, where the model of [18] is used. The comparison is done against three approaches: (1) *Chip-Level Redundant Threading (CRT)* which targets maximizing the reliability; (2) *Reliability-Timing Optimizing Technique (RTO)* jointly optimizing functional and timing dependability, but not using RMT; (3) *Timing Optimizing Technique (TO)* targeting to minimize the deadline misses. The evaluation is performed taking *TO* as a reference against which *dtune*, *CRT*, and *RTO* are compared with, where

$$\mathrm{RPF} = 100 \times \left(1 - \frac{\sum_{t \in T} \mathrm{RTP}(t)_Z}{\sum_{t \in T} \mathrm{RTP}(t)_{TO}}\right). \tag{1}$$

Figure 5a shows an overview of the achieved improvements for different chip maps with process variations, scenarios of application mixes and aging years. *dTune* achieves better RPF-results compared to *TO*, *CRT*, and *RTO* for both aging years, as it jointly considers functional and timing dependability as well as the performance variation of the cores. For year 5, a wider spread of RPF-results is observed due to the decrease in processing capabilities of the chips. Figure 5b details the application workload, where it can be observed that *CRT* performs as good as *dTune* for a lower number of applications, but does not deal well with a higher number of applications due to focusing only on minimizing functional dependability.

The solution discussed above can further be enhanced by starting with a preprocessing for application version selection, as demonstrated in [5]. First, the version with the minimal reliability penalty achieving the tolerable miss rate (for applications not being protected by RMT) and the best performance (for applications protected with RMT) are selected. Afterwards, the application-to-core mapping problem is solved for the applications protected with RMT by assigning each of them the lowest-frequency group of cores possible. Then, the applications that are not protected with RMT are mapped to cores by transforming the problem to a minimum weight perfect bipartite matching problem, which is solved by applying the *Hungarian Algorithm* [16]. The decision whether to activate RMT or not is made by iteratively adapting the mode using a heuristic in combination with the application mapping approaches.

Nevertheless, solely adopting CRT to maximize the reliability is not good enough, since the utilization of the dedicated cores may be unnecessarily low due to low utilization tasks. If the number of redundant cores is limited, the number of tasks activating RMT is also limited. When the considered multi-core systems have multi-tasking cores rather than single thread-per-core (but homogeneous performance), the same studied problems, i.e., the activation of RMT, mapping of threads to cores, and reliable code version selection, can be addressed more nicely while satisfying the hard real-time constraints. The main idea is to *use Simultaneous Redundant Threading (SRT) and CRT at the same time or even a mixture of them called Mixed Redundant Threading (MRT)*. There are six redundancy levels characterized as a set of directed acyclic graphs (DAGs) in Fig. 6, where each node (sub-task) represents a sequence of instructions and each edge represents execution dependencies between nodes.

For determining the optimal selection of redundancy levels for all tasks, several dynamic programming algorithms are proposed in [8] to provide coarse- or fine-grained selection approaches while satisfying the feasibility under *Federated Scheduling*. In extensive experiments, the proposed approaches can generally outperform the greedy approach used in *dTune* when the number of available cores is too limited to activate CRT for all tasks. Since the fine-grained approach has more flexibility to harden tasks in stage-level, the decrease of the system reliability penalty is at least as good as for the coarse-grained approach. When the resources are more limited, e.g., less number of cores, the benefit of adopting the fine-grained approach is more significant.

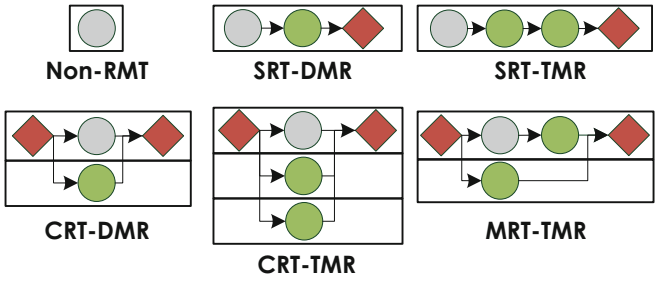


Fig. 6 DAG abstractions of the different redundancy levels, where the gray nodes are original executions and the green nodes are replicas. The red nodes represent the workload due to the necessary steps for forking the original executions and replicas, joining, and comparing the delivered results from DMR/TMR at the end of redundant multithreading. The directed edges represent the dependencies between nodes. Each block represents one core, i.e., the number of cores differs depending on the redundancy level

5 Resilient Design for System Software

Considering the adoption of error detection and recovery mechanisms due to the occurrence of soft errors from time to time, resilient designs for system software can be developed. (1) Execution versions can be determined to handle soft errors without over-provision while satisfying given robustness and timing constraints. (2) Dynamic timing guarantees can be provided without any online adaptation after a fault occurred. (3) Probabilistic analyses on deadline misses for soft real-time system. The detailed designs are presented in the following.

5.1 Adaptive Soft Error Handling

To avoid catastrophic events like unrecoverable system failures, software-based fault-tolerance techniques have the advantages in both the flexibility and application-specific assignment of techniques as well as in the non-requirement for specialized hardware. However, the main expenditure is the significant amount of time due to the additional computation incurred by such methods, e.g., redundant executions and majority voting, by which the designed system may not be feasible due to the overloaded execution demand. *Due to the potential inherent safety margins and noise tolerance, control applications might be able to tolerate a limited number of errors and only degrade its control performance.* Therefore, costly deploying full error detection and correction on each task instance might not be necessary.

To satisfy the minimal requirement of functional correctness for such control applications, (m, k) robustness constraint is proposed, which requires m out of any k consecutive instances to be correct. For each task an individual (m, k) constraint

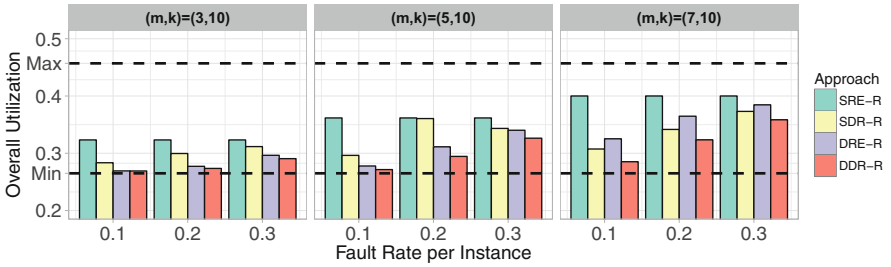


Fig. 7 Overall utilization after applying different compensation approaches on Task Path, where lower is better. Two horizontal and dashed bars represent the maximum (0.457) and the minimum utilization (0.265)

is possible to be given by other means analytically or empirically [35]. Without skipping any instances so likely achieving higher control performance, a static pattern-based approach [4] can be used to comply the reliable executions on the marked instances by following an (m, k)-pattern repeatedly to satisfy the given minimal requirement. To validate the schedulability, the multi-frame task model can then be applied to provide a hard real-time guarantee offline. A run-time adaptive approach [4] can further decide the executing version on the fly by enhancing the static pattern-based approach and monitoring the current tolerance status with sporadic replenishment counters. It is worth noting that the resulting distribution of execution jobs can still follow the (m, k) static patterns even in the worst case. Hence, the schedulability test for the static pattern-based approach can be directly used for the run-time adaptive approach as well.

Figure 7 shows the results for a self-balancing control application under different (m, k) requirements and varying fault rates. When the fault rate increases, the overall utilization of the run-time adaptive approach (DRE and DDR) also rises, since the requirement of reliable executions is increased within the application execution. Furthermore, the static pattern-based approaches (SRE and SDR) are always constant for a fixed (m, k) requirement, as the overall utilization is deterministic by the amount of job partitions. When the fault rate is as low as 10% and the (m, k) requirement is loose as $(3, 10)$, the probability of activating reliable executions is rare, and, hence, the run-time adaptive approach can closely achieve the minimum overall utilization. Overall, the results suggest that the proposed approaches can be used to serve various applications with inherent fault-tolerance depending on their perspectives, thus *avoiding over-provision under robustness and hard real-time constraints*.

5.2 Dynamic Real-Time Guarantees

When soft errors are detected, the execution time of a real-time task can be increased due to potential recovery operations. Such recovery routines may make the system

very vulnerable with respect to meeting hard real-time deadlines. This problem is often addressed by aborting *not so important* tasks to guarantee the response time of the *more important* tasks. However, for most systems such faults occur rarely and the results of *not so important* tasks might still be useful, even if they are a bit late. This implicates to not abort these not so important tasks but keep them running even if faults occur, provided that the more important tasks still meet their hard real-time deadlines. To model this behavior, the idea of *Systems with Dynamic Real-Time Guarantees* [33] is proposed, which determines if the system can provide without any online adaptation after a fault occurred, either full timing guarantees or limited timing guarantees. Please note that, this study is highly linked to the topic of mixed-criticality systems [2]. We can imagine that the system is in the low-criticality mode if *full timing guarantees* are needed, and in the high-criticality mode if only *limited timing guarantees* are provided. However, in most of the related works, such mode changes are assumed to be known, without identifying the mode change. The system only switches from low-criticality to high-criticality mode once, without ever returning to the low-criticality mode. Moreover, the low-criticality tasks are considered to be either ignored, skipped, or run with best efforts as background tasks. Such a model has received criticism as system engineers claim that it does not match their expectations in Esper et al. [11], Ernst and Di Natale [10], and Burns and Davis [2].

Suppose that a task set can be partitioned into two subsets for *more important* and *not so important* tasks, and a fixed priority order is given. To test the schedulability of a preemptive task set with constrained deadlines under a fixed priority assignment, the typical **Time Demand Analysis (TDA)** as an exact test with *pseudo-polynomial run-time* can be directly applied. To determine the schedulability for a *System with Dynamic Real-Time Guarantees*, the following three conditions must hold:

- Full timing guarantees hold, if the given task set can be scheduled according to TDA when all tasks are executed in the normal mode.
- When the system runs with limited timing guarantees, all *more important* tasks will meet their deadlines if they can be proven to be scheduled by TDA while all tasks are executed in the abnormal mode.
- Each *not so important* task has bounded tardiness if the sum of utilization over all tasks in the abnormal mode can be less than or equal to one.

To decide such a fixed priority ordering for a given task set, the **Optimal Priority Assignment (OPA)** can be applied to find a feasible fixed priority assignment, since the above schedulability test is OPA compatible. It is proven that a feasible priority assignment for a *System with Dynamic Real-Time Guarantees* can be found if one exists by using the priority assignment algorithm presented in [33], which has a much better run-time than directly applying OPA.

As faulty-aware system design is desirable in the industrial practice, having an online monitor to reflect the system status is also important. This monitor should trigger warnings if the system can only provide limited timing guarantees, and display the next time the system will return to full timing guarantees. To achieve

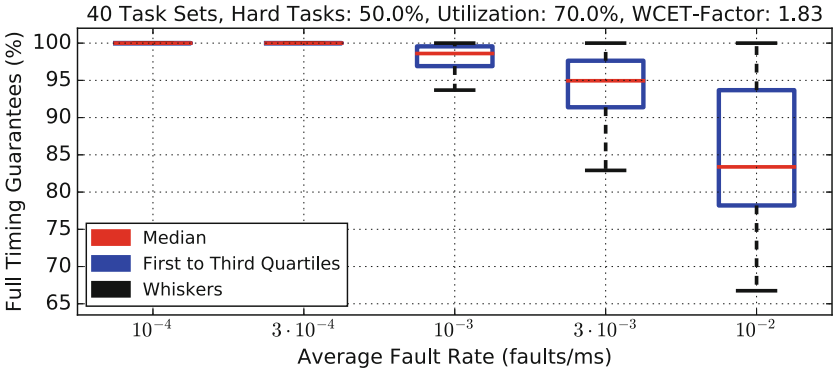


Fig. 8 Percentage of Time where *Full Timing Guarantees* can be given for task sets with utilization 70% in the normal mode under different fault rates. The median of the acceptance rates over 40 task sets is colored in red. The blue box represents the interval around this median that contains the inner 50% of those values while the whiskers display the range of the top/bottom 25% of those values

this, an approximation is needed to detect the change from *full timing guarantees* to *limited timing guarantees*, and for the calculation of an upper bound of the next time instance the system will return to full timing guarantees. To realize the routine of the online monitor, the system software has to ensure that the release pattern is still correct when a task misses its deadline and there is a helper function to keep tracking the number of postponed releases. How to enhance a real-time operating system for the previous two requirements is further discussed in [6].

Figure 8 shows the results with the percentage of time that the system was running with *full timing guarantees*. At a fault rate of 10^{-4} and 3×10^{-4} (faults/ms), the system always provides *full timing guarantees*. When the fault rate is increased, the average of the time where *full timing guarantees* are provided drops. For the worst-case values, the drop is faster but even in this case full timing guarantees are still provided $\approx$92.59% and $\approx$82.91% of the time for fault rates of 10^{-3} and 3×10^{-3}, respectively. This shows that even for the higher fault rates under a difficult setting, the system is still able to provide full timing guarantees for a reasonable percentage of time.

5.3 Probabilistic Deadline-Miss Analyses

When applying software fault-tolerant techniques, one natural assumption is that the *system functions normally most of time*. Therefore, it is meaningful to model the occurrence of different execution of a task by *probabilistic bounds on the worst-case execution time (WCETs)* due to potential recovery routines. This allows the system designer to provide probabilistic arguments, e.g., *Deadline-Miss Probability (DMP)*

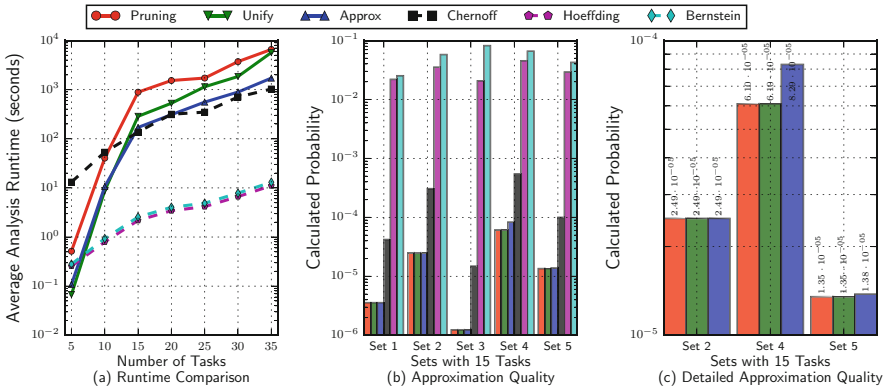


Fig. 9 (**a**) Average run-time with respect to task set cardinality. (**b**) Approximation quality for five task sets with Cardinality 15. (**c**) Detailed approximation quality for the convolution-based approaches

and *Deadline-Miss Rate*, as the statistical quantification to evaluate the proposed analyses scheduling algorithms, etc.

To derive the DMP, statistical approaches, i.e., *Probabilistic response time analysis* and *Deadline-misses probability analysis*, are usually taken into consideration. The state of the art of the probabilistic response time analysis is based on task-level convolution-based approaches [34]. Naturally, convolution-based approaches are computationally expensive to be applied when the number of tasks or jobs is large. Alternatively, *Deadline-Misses probability analysis* [3] is proposed, which can utilize analytical bounds, e.g., *Chernoff bounds* [3, 9], *Hoeffding's and Bernstein's inequalities* [34]. Please note that, the deadline-misses probability analysis is not better than the probabilistic response time analysis in terms of accuracy of the DMP. However, it is essentially much faster and has a better applicability in practice.

Figure 9 shows the results for randomly generated tasks sets with a normal-mode utilization 70%, fault rate 0.025, and for all tasks the execution time of abnormal mode is assumed to be two times of the normal mode. Three approaches based on the task-level convolution-based approaches [34], i.e., *Pruning, Unify, Approx*, result in similar values, roughly one order of magnitude better than *Chernoff* [3]. Although *Bernstein* [34] and *Hoeffding* [34] are orders of magnitude faster than the other approaches which are compatible with respect to the related run-time, the error of them is large compared to *Chernoff* by several orders of magnitude. The results suggest that, if sufficiently low deadline-miss probability can be guaranteed from analytical bounds, the task-level convolution-based approach then can be considered.

DMP and Deadline-Miss Rate are both important performance indicators to evaluate the extent of requirements compliance for soft real-time systems. However, the aforementioned probabilistic approaches all focus on finding the probability of the first deadline miss, and it is assumed that after a deadline miss the system either

discards the job missing its deadline or reboots itself. Therefore, the probability of one deadline miss directly relates to the deadline-miss rate since all jobs can be considered individually. If this assumption do not hold, the additional workload due to a deadline miss may trigger further deadline misses.

To derive a tight but safe estimation of the deadline-miss rate, an event-driven simulator [7] with a fault injection module can be used, which can gather deadline-miss rates empirically. However, the amount of time needed to perform the simulations is too large. Instead of simulating the targeted task set, an analytical approach [7] can leverage on the above probabilistic approaches that over-approximate the DMP of individual jobs to derive a safe upper bound on the expected deadline-miss rate.

6 Application-Specific Dependability

In this section, we focus on application-specific aspects on dependability improvement with the help of a case study on the *Context Adaptive Variable Length Coding (CAVLC)* used in the H.264 video coding standard [20, 30, 32]. It summarizes *how application-specific knowledge can be leveraged to design a power-efficient fault-tolerance technique for H.264 CAVLC.*

CAVLC is an important part of the coding process and is susceptible to errors due to its context adaptivity, multiple coding tables, and complex structure. It transforms an input with a fixed length to flexible-length code consisting of *codeword/codelength* tuples. The impact of a single error on the subjective video quality is illustrated in Fig. 10a, which shows a significant distortion in a video frame when the header of a macroblock (i.e., a 16×16 pixels block) is affected. Faults during the CAVLC can also propagate to subsequent frames or even lead to encoder/decoder crashes.

Consequently, it is required to address these problems during the CAVLC execution. To reduce the overhead compared to generic solutions, application-

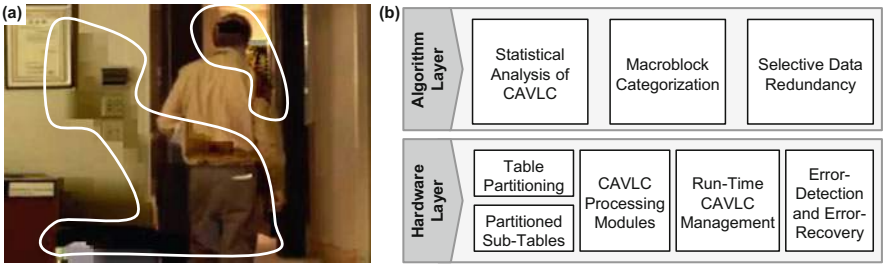


Fig. 10 (**a**) Example of a corrupted frame showing the effects of a single-bit error. (**b**) Overview of the contributions for the dependable CAVLC and the corresponding system layers (adapted from [32])

specific knowledge is considered. Specifically, Fig. 10b shows an overview of the dependable CAVLC with contributions on the architecture and algorithm/software layer, which are based on exploiting the video content properties and performing a statistical analysis of CAVLC.

- **Application-Specific Knowledge** is considered by (1) an analysis of error probabilities, (2) distribution of different syntax elements, (3) algorithmic properties, and (4) specifications defined by the standard. It includes an analysis of different macroblock categories (homogeneous/textured, fast/slow motion). The most important observations are that the *total non-zero coefficients have a significant influence on the error probabilities of different syntax elements*. They can be used to detect potential errors at the algorithm level if the macroblock properties are known.
- **Selective Data Redundancy:** Based on the application-specific knowledge obtained by the analysis, selected CAVLC data (e.g., quantized coefficients, coefficient statistics, etc.) can be protected by storing redundant copies and parity data in unused data structures. This is possible, e.g., for the quantized coefficients as the quantization often leads to unused ("0") entries, where redundant data can be stored in a reflected fashion. *Only the low-frequency coefficients are protected in case the space is insufficient.*
- **Dependable CAVLC Hardware Architecture:** The original and redundant values are loaded by a hardware module, which performs error detection and error recovery. In case of a mismatch, the parity is calculated and compared to the stored one, so that the correct entry can be found. A recovery is even possible if both entries are corrupted by reloading the original block and performing the quantization step again. Additionally, the coding tables used by CAVLC for obtaining the *codeword* and *codelength* need to be protected. For that, the individual tables are split into different sub-tables, where the partitioning decision is based on the distribution of the syntax elements. *Sub-tables not being accessed frequently can then be power-gated for leakage energy savings.* For each sub-table, a block parity-based protection approach is used for error detection, trading-off the additional memory required and the protection offered. Furthermore, entries not being accessed due to the algorithm properties and zero-entries are not stored. Similarly, *the data in tables containing mirrored entries also has to be stored only once*, thereby further reducing the memory requirements and leakage energy.
- **Run-Time Manager:** The dependable CAVLC architecture is controlled by a run-time manager which activates/deactivates the power-gating of the memory parts storing the sub-tables, loads the requested data from the tables, and controls error detection and reloading of data.
- **Dependable CAVLC Processing Flow:** The overall flow starts with a macroblock characterization, which determines the power-gating decision. Then, highly probable values for the syntax elements are predicted, which are used later for the algorithm-guided error detection. Afterwards, the header elements

are loaded by the hardware module performing error detection and error recovery. Finally, the quantized coefficients are coded by CAVLC for each 4×4 block.

This example architecture illustrates how application-specific knowledge can be leveraged to improve the design decisions for enhancing the dependability of the system and its power consumption. It achieves significant improvements in terms of the resulting video quality compared to an unprotected scheme. Moreover, *leakage energy savings of* 58% *can be achieved by the application-guided fault-tolerance and table partitioning.*

7 Conclusion

Dependability has emerged as an important design constraint in modern computing systems. For a cost-effective implementation, a cross-layer approach is required, which enables each layer to contribute its advantages for dependability enhancement. This chapter presented contributions focusing on the architecture, SW/OS, and application layers. Those include modeling and estimation techniques considering functional correctness and timeliness of applications as well as approaches for generating dependable software (e.g., by dependability-aware software transformations or selective instruction redundancy). Additionally, the run-time system is employed for selecting appropriate dependable application versions and adapting to different workloads and run-time conditions, enabling a tradeoff between performance and dependability. It has furthermore been shown how application-specific characteristics can be used to enhance the dependability of a system, taking the example of a multimedia application.

Acknowledgments This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500—spp1500.itec.kit.edu).

References

1. Alam, M.A., Kufluoglu, H., Varghese, D., Mahapatra, S.: A comprehensive model for PMOS NBTI degradation: recent progress. Microelectron. Reliab. **47**(6), 853–862 (2007). https://doi.org/10.1016/j.microrel.2006.10.012
2. Burns, A., Davis, R.: Mixed criticality systems-a review, 7th edn. Tech. rep., University of York (2016)
3. Chen, K., Chen, J.: Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In: 12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, 14–16 June 2017, pp. 1–8 (2017). https://doi.org/10.1109/SIES.2017.7993392

4. Chen, K., Bönninghoff, B., Chen, J., Marwedel, P.: Compensate or ignore? Meeting control robustness requirements through adaptive soft-error handling. In: Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Santa Barbara, 13–14 June 2016, pp. 82–91 (2016). https://doi.org/10.1145/2907950.2907952
5. Chen, K., Chen, J., Kriebel, F., Rehman, S., Shafique, M., Henkel, J.: Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity. IEEE Trans. Comput. **65**(11), 3441–3455 (2016). https://doi.org/10.1109/TC.2016.2532862
6. Chen, K.H., Von Der Brüggen, G., Chen, J.J.: Overrun handling for mixed-criticality support in RTEMS. In: WMC 2016, Proceedings of WMC 2016, Porto (2016). https://hal.archives-ouvertes.fr/hal-01438843
7. Chen, K., von der Brüggen, G., Chen, J.: Analysis of deadline miss rates for uniprocessor fixed-priority scheduling. In: 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018, Hakodate, 28–31 August 2018, pp. 168–178 (2018). https://doi.org/10.1109/RTCSA.2018.00028
8. Chen, K., von der Bruggen, G., Chen, J.: Reliability optimization on multi-core systems with multi-tasking and redundant multi-threading. IEEE Trans. Comput. **67**(4), 484–497 (2018). https://doi.org/10.1109/TC.2017.2769044
9. Chen, K.H., Ueter, N., von der Brüggen, G., Chen, J.: Efficient computation of deadline-miss probability and potential pitfalls. In: Design, Automation and Test in Europe, DATE 19, Florence, 25–29 March 2019
10. Ernst, R., Di Natale, M.: Mixed criticality systems - a history of misconceptions? IEEE Des. Test **33**(5), 65–74 (2016). https://doi.org/10.1109/MDAT.2016.2594790
11. Esper, A., Nelissen, G., Nélis, V., Tovar, E.: How realistic is the mixed-criticality real-time system model? In: RTNS, pp. 139–148 (2015)
12. 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 Integr. Circuits Syst. **32**(1), 8–23 (2013). https://doi.org/10.1109/TCAD.2012.2223467
13. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization, 2001, WWC-4. 2001 IEEE International Workshop, WWC '01, pp. 3–14. IEEE Computer Society, Washington (2001). https://doi.org/10.1109/WWC.2001.15
14. 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, 9–14 October 2011, pp. 69–78 (2011). https://doi.org/10.1145/2039370.2039384
15. Henkel, J., Bauer, L., Dutt, N.D., Gupta, P., Nassif, S.R., Shafique, M., Tahoori, M.B., Wehn, N.: Reliable on-chip systems in the nano-era: lessons learnt and future trends. In: The 50th Annual Design Automation Conference 2013, DAC '13, Austin, 29 May–07 June 2013, pp. 99:1–99:10 (2013). https://doi.org/10.1145/2463209.2488857
16. Kuhn, H.: The Hungarian method for the assignment problem. Naval Res. Logist. Quart. **2**, 83–98 (1955). https://doi.org/10.1002/nav.20053
17. Oh, N., Shirvani, P.P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. IEEE Trans. Reliab. **51**(1), 63–75 (2002). https://doi.org/10.1109/24.994913
18. Raghunathan, B., Turakhia, Y., Garg, S., Marculescu, D.: Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In: Design, Automation and Test in Europe, DATE 13, Grenoble, 18–22 March 2013, pp. 39–44 (2013). https://doi.org/10.7873/DATE.2013.023
19. 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, 9–14 October 2011, pp. 237–246 (2011). https://doi.org/10.1145/2039370.2039408
20. Rehman, S., Shafique, M., Kriebel, F., Henkel, J.: Revc: computationally reliable video coding on unreliable hardware platforms: a case study on error-tolerant H.264/AVC CAVLC entropy coding. In: 18th IEEE International Conference on Image Processing, ICIP 2011, Brussels, 11–14 September 2011, pp. 397–400 (2011). https://doi.org/10.1109/ICIP.2011.6116533
21. Rehman, S., Shafique, M., Kriebel, F., Henkel, J.: RAISE: reliability-aware instruction scheduling for unreliable hardware. In: Proceedings of the 17th Asia and South Pacific Design Automation Conference, ASP-DAC 2012, Sydney, 30 January–2 February 2012, pp. 671–676 (2012). https://doi.org/10.1109/ASPDAC.2012.6165040
22. Rehman, S., Shafique, M., Henkel, J.: Instruction scheduling for reliability-aware compilation. In: The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, 3–7 June 2012, pp. 1292–1300 (2012). https://doi.org/10.1145/2228360.2228601
23. 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, 9–11 April 2013, pp. 273–282 (2013). https://doi.org/10.1109/RTAS.2013.6531099
24. 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, 18–22 March 2013, pp. 1759–1764 (2013). https://doi.org/10.7873/DATE.2013.354
25. 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
26. Rehman, S., Kriebel, F., Sun, D., Shafique, M., Henkel, J.: dtune: leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, 1–5 June 2014, pp. 84:1–84:6 (2014). https://doi.org/10.1145/2593069.2593127
27. Rehman, S., Chen, K., Kriebel, F., Toma, A., Shafique, M., Chen, J., Henkel, J.: Cross-layer software dependability on unreliable hardware. IEEE Trans. Comput. **65**(1), 80–94 (2016). https://doi.org/10.1109/TC.2015.2417554
28. Rehman, S., Shafique, M., Henkel, J.: Reliable Software for Unreliable Hardware - A Cross Layer Perspective. Springer (2016). https://doi.org/10.1007/978-3-319-25772-3
29. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Software-controlled fault tolerance. Trans. Archit. Code Optim. **2**(4), 366–396 (2005). https://doi.org/10.1145/1113841.1113843
30. Shafique, M., Zatt, B., Rehman, S., Kriebel, F., Henkel, J.: Power-efficient error-resiliency for H.264/AVC context-adaptive variable length coding. In: 2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, 12–16 March 2012, pp. 697–702 (2012). https://doi.org/10.1109/DATE.2012.6176560
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, 29 May–07 June 2013, pp. 17:1–17:9 (2013). https://doi.org/10.1145/2463209.2488755
32. Shafique, M., Rehman, S., Kriebel, F., Khan, M.U.K., Zatt, B., Subramaniyan, A., Vizzotto, B.B., Henkel, J.: Application-guided power-efficient fault tolerance for H.264 context adaptive variable length coding. IEEE Trans. Comput. **66**(4), 560–574 (2017). https://doi.org/10.1109/TC.2016.2616313
33. von der Bruggen, G., Chen, K., Huang, W., Chen, J.: Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In: 2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, 29 November–2 December 2016, pp. 303–314 (2016). https://doi.org/10.1109/RTSS.2016.037

34. von der Brüggen, G., Piatkowski, N., Chen, K., Chen, J., Morik, K.: Efficiently approximating the probability of deadline misses in real-time systems. In: 30th Euromicro Conference on Real-Time Systems, ECRTS 2018, Barcelona, 3–6 July 2018, pp. 6:1–6:22 (2018). https://doi.org/10.4230/LIPIcs.ECRTS.2018.6
35. Yayla, M., Chen, K., Chen, J.: Fault tolerance on control applications: empirical investigations of impacts from incorrect calculations. In: 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems, EITEC@CPSWeek 2018, 10 April 2018, Porto, pp. 17–24 (2018). https://doi.org/10.1109/EITEC.2018.00008

Fault-Tolerant Computing with Heterogeneous Hardening Modes

Florian Kriebel, Faiq Khalid, Bharath Srinivas Prabakaran, Semeen Rehman, and Muhammad Shafique

1 Introduction

Recent technological advancements in the field of transistor fabrication, such as FinFETs and GAAFETs, have led to significant improvements in the performance of next-generation multi-core processors but at the expense of an increased susceptibility to reliability threats such as soft errors [4, 37], aging [14], and process variations [14]. These threats generate permanent and/or temporary faults that can lead to unexpected system failures and can be disastrous to several safety-critical applications such as automotive, healthcare, aerospace, etc., as well as high-performance computing systems. Therefore, several techniques have been proposed to detect, prevent, and mitigate these reliability threats across the computing stack ranging from the transistor and circuit layer [27, 43] to the software/application layer [2, 42, 44]. Oftentimes, (full-scale) redundancy is employed at the hardware and the software layers, for example, at the *software layer*, by executing multiple redundant thread versions of an application, either spatially or temporally, and at the *hardware layer*, by duplicating or triplicating the pipeline, i.e., Double/Triple Modular Redundancy (DMR/TMR) [28, 29, 32, 46]. However, these reliability techniques exhibit several key limitations, as discussed below:

1. Ensuring temporal redundancy at the software layer, by executing multiple redundant threads of a given application on the same core, would incur a significant performance overhead.

F. Kriebel (✉) · F. Khalid · B. S. Prabakaran · S. Rehman · M. Shafique
Technische Universität Wien (TU Wien), Vienna, Austria
e-mail: florian.kriebel@tuwien.ac.at; faiq.khalid@tuwien.ac.at; bharath.prabakaran@tuwien.ac.at; semeen.rehman@tuwien.ac.at; muhammad.shafique@tuwien.ac.at

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_7

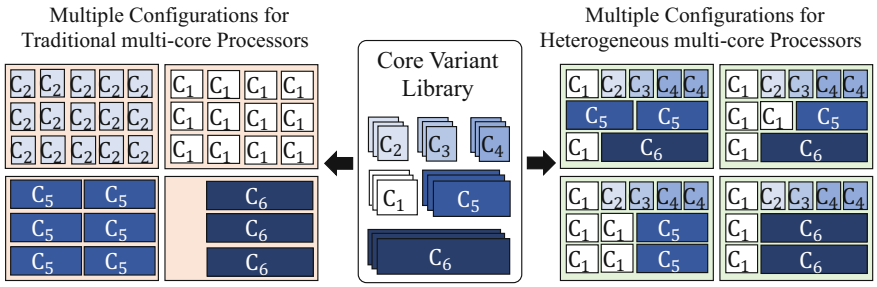


Fig. 1 Different configurations for mitigating dependability threats for traditional (homogeneous) and heterogeneous multi-core systems, respectively

2. Executing multiple redundant threads in multiple cores concurrently, instead of a single core, provides spatial redundancy and nullifies the performance overhead caused by the temporal redundancy. However, due to the activation of multiple cores, this technique incurs a significant power overhead.
3. Similarly, fabricating redundant hardware components to provide full-scale TMR across the pipeline incurs additional area, power, and energy overheads including additional on-chip resources for the data correction and control units.
4. Moreover, these techniques are not adaptive with respect to the dependability requirements of the applications, as well as their inherent error tolerance, during their execution.

To address these limitations, we proposed the **reliability-heterogeneous architectures** in [20, 21, 33, 34]. They offer different types of reliability modes in different cores (i.e., the so-called *reliability-heterogeneous cores*), realized through hardening of different pipeline components using different reliability mechanisms. Hence, such processors provide a foundation for design- and run-time trade-offs in terms of reliability, power/energy, and area. Their motivation arises from the fact that different applications exhibit varying degrees of error tolerance and inherent masking to soft errors due to data and control flow masking. Hence, depending upon the executing applications, their tasks can be mapped to a set of *reliability-heterogeneous cores* to mitigate soft errors, as shown in Fig. 1.

Although this solution significantly reduces the power/energy and performance overheads, it requires a sophisticated *run-time management system* that performs an appropriate code-to-core mapping of the applications, based on their requirements and given power/performance constraints. This requires enabling certain features across the hardware and software layers such as additional control logic, core monitoring units at the hardware layer, and a run-time manager at the software layer. Embedding this chapter's content in the scope of this book and the overall projects [11, 13], the focus of this chapter is limited to the design of such

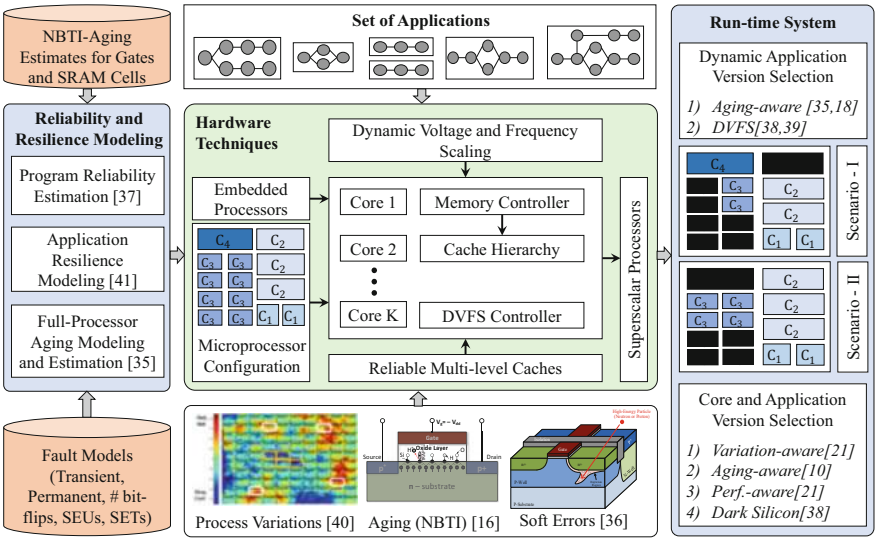


Fig. 2 An overview of dependable computing with heterogeneous hardening modes (adapted from [34]). Image sources: [16, 37]

hardware/software techniques that can enable heterogeneous dependable computing (see Fig. 3).

Typically, the hardware solutions for dependable heterogeneous architectures consist of the following three phases (see Fig. 2):

1. **Reliability and Resilience Modeling:** First, the effects of different reliability threats (i.e., soft errors, aging, and process variations) on different components of a given multi-core system and different applications are modeled and analyzed based on mathematical analysis, simulation, and/or emulation.
2. **Hardware Techniques:** Based on the vulnerability analysis of the previous step, multiple reliability-heterogeneous core variants are developed by hardening a combination of the pipeline and/or memory components. Similarly, an analysis of multi-level cache hierarchies has led to the design of multiple heterogeneous reliability cache variants and reliability-aware reconfigurable caches.
3. **Run-time System:** Afterwards, appropriate task-to-core mapping as well as reliable code version selection are performed, while satisfying the application's reliability requirement and minimizing the power/area overheads. These problems can also be formulated as constrained optimization problems.

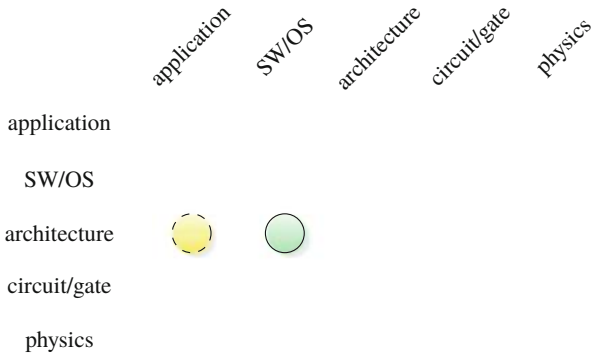


Fig. 3 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

2 Fault-Tolerant Heterogeneous Processors

Reliability threats not only affect the computing cores in the microprocessors, but can also significantly affect the on-chip memory sub-systems, like multi-level caches. This section provides an overview of our techniques for developing reliability-heterogeneous in-order processors and multi-level cache hierarchies. Unlike the traditional homogeneous dependable processors, the development of reliability-heterogeneous processors not only requires design-time efforts to develop multiple variable-reliability processor variants but also requires a run-time management system that can efficiently cater the applications' requirements (see Fig. 2). Therefore, as illustrated in Fig. 4, developing these hardware techniques can be divided into two phases, namely, design-time and run-time:

1. **Design-Time:** At design-time, first the overall vulnerability of a processor is analyzed. Based on this analysis, we develop hardware techniques that can be used to design reliability-heterogeneous processor cores (see Sect. 2.1). In the next step, these hardened cores are integrated into an architectural-level simulator to evaluate their effectiveness. Similarly, we evaluate the vulnerability of caches, based on which hardware techniques are designed to mitigate the effects of reliability threats in caches (see Sect. 2.2). These reliability-aware caches and multiple reliability-heterogeneous cores are used to design a reliability-heterogeneous processor, as depicted by *Design-Time* in Fig. 4.
2. **Run-time:** To effectively use the reliability-heterogeneous processor, an *adaptive run-time manager for soft error resilience (ASER)* is used to estimate the reliability requirements of the applications (as well as their resilience properties), and to efficiently map their threads to a set of hardened cores while adhering to the user and performance constraints, as depicted by *Run-Time* in Fig. 4.

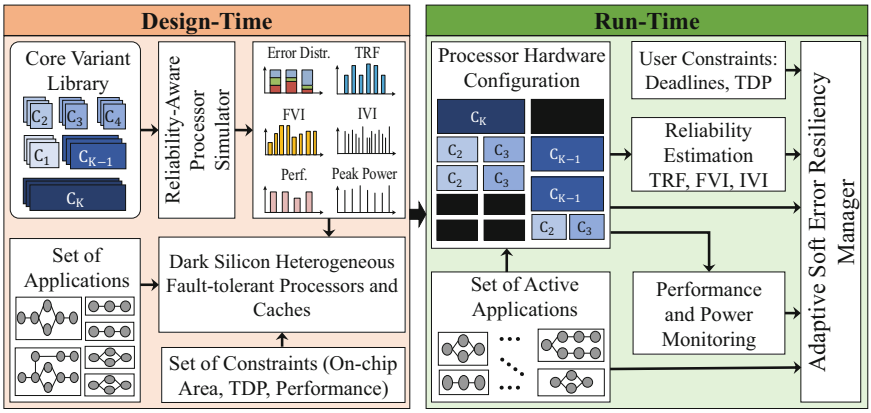


Fig. 4 The design- and run-time methodology to develop dependable heterogeneous processors (adapted from [20, 34])

2.1 Hardening Embedded Processors

To design the hardened cores for reliability-heterogeneous architectures, first, we analyze the vulnerability of these cores to different reliability threats. Based on this vulnerability analysis, instead of enabling full-scale DMR/TMR, we design the micro-architecture of different hardened (in-order) cores. These cores have distinct reliability mechanisms in different pipeline components (ranging from unprotected to fully-protected), but implement the same instruction set architecture (ISA); see the core variant library in Fig. 4. Hence, these cores provide a trade-off between reliability, area, and power/energy consumption. Since not all transistors on a chip can be powered-on at the same time (i.e., the dark silicon problem [7, 31, 41]), we leverage this fact to integrate many different hardened cores to develop a reliability-heterogeneous ISO-ISA processor [20, 21], while adhering to hardware and user-defined constraints (e.g., area, power) considering a target domain (i.e., given a particular set of target applications).

To cater for the application-specific requirements at run-time, an *adaptive run-time manager for soft error resilience (ASER)* determines an efficient application-to-core mapping considering the application's vulnerability and deadline requirements, system performance, thermal design power (TDP), and other user-defined constraints. For example, Fig. 5 depicts the varying reliability improvements of the ASER run-time system approach in comparison with multiple state-of-the-art reliability techniques such as *TRO* (timing dependability optimization aiming at minimizing the deadline misses), *RTO* (optimizing functional as well as timing dependability), *Full-TMR* (activating full TMR), and *AdTMR* (deactivating TMR when the vulnerability lies below a pre-defined threshold). The reliability is measured using the Reliability Profit Function (RPF) which is defined as follows:

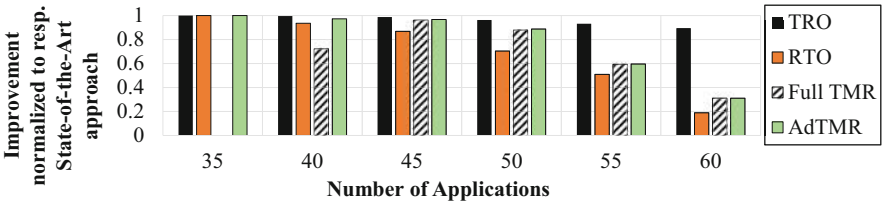


Fig. 5 Reliability Profit Function improvement over different state-of-the-art reliability techniques, i.e., timing reliability optimization (TRO), functional and timing reliability (RTO), TMR, adaptive TMR (adTMR) (adapted from [20])

$$RPF = 1 - \left(\frac{RTP(t)_{ASER}}{RTP(t)_{Z}} \right) \tag{1}$$

where $\forall t \in T$, T is a set of run-time concurrently executing application tasks ($T = \{T_1, T_2, \ldots, T_M)\}$), $Z \in \{TRO, RTO, TMR, adTMR\}$, and RTP is the Reliability-Timing Penalty [38]. Note, a higher value of RPF translates to a better reliability. The ASER approach achieves 58–96% overall system reliability improvements when compared to these four state-of-the-art techniques.

2.2 Reliability Techniques for Multi-Level Caches

In any microprocessor, on-chip memories play a significant role to improve the throughput and performance of an application. Moreover, memory elements (such as caches) are even more susceptible to soft errors compared to the computing elements (i.e., logic) as they occupy a significant portion of the total on-chip area [9]. Therefore, for designing dependable multi/many-core processors, different (individual) cache levels as well as the complete cache hierarchy (considering inter-dependency between different cache levels) have to be analyzed and optimized for mitigating reliability threats.

2.2.1 Improving the Reliability of Last-Level Caches

Dynamic reconfiguration of the caches with respect to the running applications has a significant impact on the vulnerability of the on-chip last-level caches, as shown in Fig. 6. It can be observed from the vulnerability analysis of a given cache configuration (see Fig. 6) that due to different access patterns and occupancy of last-level caches for the application, the vulnerability also varies depending on the executing applications. This dynamic change in vulnerability at run-time can be exploited to improve the reliability of the last-level cache. Therefore, dynamic reconfiguration of the last-level cache is exploited to develop a *reliability-aware*

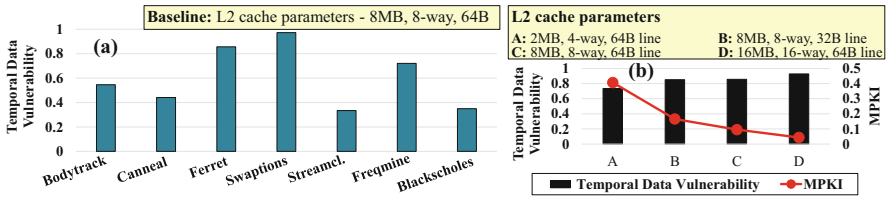


Fig. 6 (**a**) Vulnerability analysis of different applications from the *PARSEC* benchmark for the baseline case (L2 cache parameters—8 MB, 8-way, 64 B). (**b**) Vulnerabilities and cache misses (MKPI) for the *Ferret* application for different cache configurations (adapted from [19])

reconfigurable cache architecture [19, 22]. Towards this, we aim at reducing the vulnerability of concurrently executing applications by employing the following features:

1. A methodology to *quantify the cache vulnerability* with respect to concurrently executing applications.
2. A method for *lightweight online prediction of the application vulnerability online* based on the cache utilization and performance data.
3. A methodology to *dynamically reconfigure the last-level cache* at run-time that targets at minimizing the application vulnerability w.r.t. cache while keeping the performance overhead low, or within a tolerable bound.

This reliability-aware cache reconfiguration [22] can also be applied in conjunction with the error correcting codes (ECCs). For example, Single Error Correcting-Double Error Detecting (SEC-DED) [6] can be combined with the reliability-aware cache reconfiguration [22] to improve reliability in multi-bit error scenarios, or in cases where only some of the cache partitions are ECC-protected due to the area constraints.

2.2.2 Improving the Reliability of the Complete Cache Hierarchy

The application vulnerability towards soft errors is not only dependent on the individual utilization or dynamic reconfiguration of the different individual cache levels (e.g., L1 or L2). Rather, the *vulnerability interdependencies across different cache* levels also have significant impact on the reliability of the system. Therefore, the vulnerability of the concurrently executing applications with respect to the corresponding cache configuration can further be improved by considering these interdependencies across different cache levels. To achieve an efficient design, we first performed an *architectural design space exploration* (DSE), while considering multi-core processors with multiple cache levels executing different multi-threaded applications. Our cache DSE methodology identifies the pareto-optimal configurations with respect to constraints, performance overhead, and targeted vulnerabilities [45]. Afterwards, these configurations are used at run time to

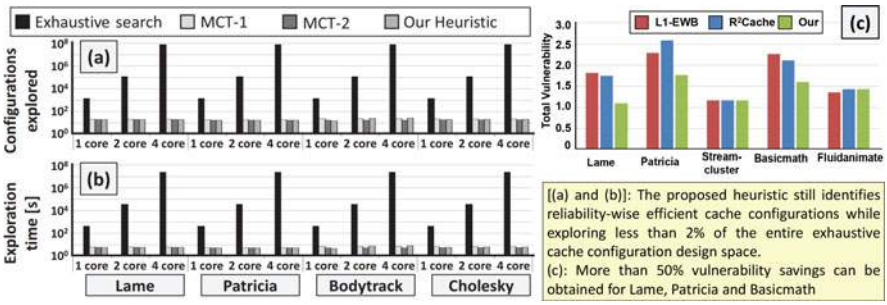


Fig. 7 (**a**) and (**b**) Exploration time saving achieved by the proposed approach with respect to exhaustive exploration, multi-level tuning approach (MCT-1) [47] and heuristic (MCT-2) [47]. (**c**) Vulnerability saving comparison of the proposed approach compared to non-reconfigurable baseline cache with L1 Early WriteBack (EWB) [15] and reliability-aware last-level cache partitioning (R^2Cache) [22] schemes (adapted from [45])

perform *reliability-aware cache reconfiguration for the complete cache hierarchy*. Figure 7 shows that more than 50% vulnerability saving is achieved by the proposed solution as compared to non-reconfigurable baseline cache with L1 Early WriteBack (EWB) [15] and Reliability-Aware Last-Level Cache Partitioning (R^2Cache) while exploring less than 2% of the entire exhaustive cache configuration design space.

3 Heterogeneous Reliability Modes of Out-of-Order Superscalar Cores

Embedded processors, although important in a wide range of applications and scenarios, cannot cater the high throughput and performance requirements of personal computers or high-performance computing platforms such as cloud servers or data-centers, which are also constrained in the amount of power that can be consumed. Such high-throughput systems deploy multi-core out-of-order (O3) superscalar processors, such as Intel Core i7 processors in PCs, and Intel Xeon or AMD Opteron processors in servers and data-centers worldwide. An O3 processor executes the instructions of a program *out-of-order*, instead of in-order as is the case in embedded processors (e.g., LEON3), to utilize the instruction cycles that would otherwise be wasted in pipeline stalls. A *superscalar* processor, on the other hand, implements instruction-level parallelism to execute more than one instruction in parallel by dispatching instructions to multiple different execution units embedded in the processor core. Therefore, an O3 superscalar processor offers a significantly higher throughput by combining the advantages of these two individual techniques. However, enabling such high throughput comes at the cost of implementing additional hardware units such as the Re-order Buffer (ROB), which keeps track of the instructions executing out-of-order.

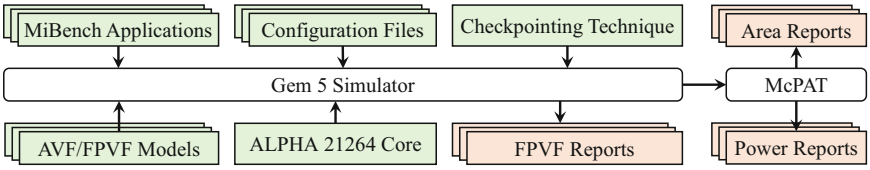


Fig. 8 Experimental tool-flow for vulnerability analysis of out-of-order superscalar *ALPHA* cores

In this section, we analyze the vulnerability of the ALPHA 21264 [17] O3 superscalar processor and design multiple reliability-heterogeneous processor cores from which an optimal configuration can be chosen at run-time based on the applications' reliability requirements.

3.1 Experimental Setup

Figure 8 presents an overview of the tool-flow used to obtain the results. We utilize a modified version of the *gem5 simulator* [5] extended to support the following functionality:

1. Determine the *vulnerable time* of all pipeline components, which in turn is used to compute their *Architectural Vulnerability Factors (AVFs)* [30],
2. *Full support for simulating reliability-heterogeneous cores* obtained by triplicating key pipeline components (instead of implementing full-scale TMR), and
3. *Checkpoint processor state compression* using techniques like DMTCP [3], HBICT [1], and GNU zip [8].

We evaluate our reliability-heterogeneous *ALPHA* 21264 four-issue superscalar processor cores using the *MiBench* application benchmark suite [12].

3.2 Vulnerability Analysis of Out-of-Order Superscalar Processors

The AVF of a component C over a period of N clock-cycles is defined as the probability of a fault that is generated in C to propagate to the final output resulting in an erroneous application output or intermittent termination of the program [30]. It is computed using the following equation:

$$AVF_C = \frac{\sum_{n=0}^{n=N} VulnerableBits}{TotalBits \times N} \tag{2}$$

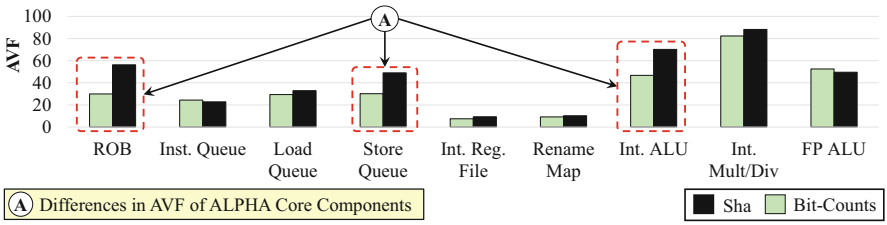


Fig. 9 Differences in AVF of *ALPHA* core components during application execution (*SHA* and *Bit-counts*) (adapted from [33])

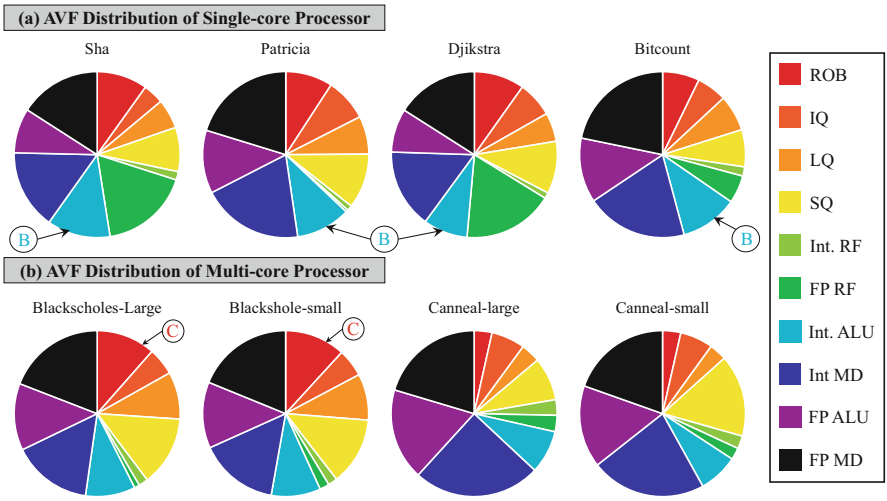


Fig. 10 Vulnerability analysis of *ALPHA* cores in single- and multi-core processors (adapted from [33])

The AVF of each pipeline component is estimated using applications from the *MiBench* and *PARSEC* application benchmark suites for single- and multi-core *ALPHA* 21264 superscalar processors for key pipeline components such as: (1) Re-order Buffer (ROB), (2) Instruction (IQ), (3) Load (LQ), (4) Store Queues (SQ), (5) Integer Register Files (Int. RF), (6) Floating Point Register Files (FP RF), (7) Rename Map (RM), (8) Integer ALUs (Int. ALU), (9) Floating Point ALUs (FP ALU), (10) Integer Multiply/Divide (Int. MD), and (11) Floating Point Multiply/Divide (FP MD). Figures 9 and 10 illustrate the results of the vulnerability analysis experiments for both the single-core and multi-core processors.

We analyze the results obtained from the vulnerability analysis to make the following **key observations**:

1. We have identified three key pipeline components (Integer ALU, Store Queue, and Re-order Buffer) that are more vulnerable during the execution of *SHA*, when compared to *Bit-counts*, as depicted by *A* in Fig. 9.

2. The AVFs of the individual pipeline components vary for different application workloads. For example, as shown in Fig. 10a the vulnerability of the Integer ALU widely varies for the four application workloads evaluated (labeled *B*).
3. In case of multi-core processors, the size of the input data does not significantly affect the AVF of the pipeline components, as shown by *C* in Fig. 10b.

The AVF of a component varies based on the type and number of instructions present in the application and its properties such as its compute- or memory-intensiveness, instruction-level parallelism, cache hit/miss rate, etc. For example, components like the ROB and the SQ are more vulnerable in *SHA* because of higher levels of instruction-level parallelism and more store instructions.

Therefore, based on this information, we can select certain key pipeline components that can be hardened/triplicated to increase the reliability of the processor for a given application workload. By hardening multiple key pipeline components in different combinations, we design a wide range of reliability-heterogeneous O3 superscalar *ALPHA* cores from which an optimal design configuration can be selected at run-time based on an application's reliability requirement while minimizing the area and/or power overheads.

3.3 Methodology for Hardening Out-of-Order Superscalar Processors

Our methodology for designing reliability-heterogeneous O3 superscalar processors targets two key approaches: (1) Redundancy, and (2) Checkpointing. Redundancy at the hardware layer is ensured by designing a wide range of reliability-heterogeneous processor cores by hardening a combination of the vulnerable pipeline components, depending on the reliability requirements of the target application. The vulnerable components are selected based on the fault-injection experiments and the AVF values of each component for different application workloads. Second, to further enhance processor reliability, we investigate and analyze various compression mechanisms that can be used to efficiently reduce the size of checkpointing data. An overview of our methodology for hardening O3 superscalar processors is presented in Fig. 11. First, we explain how we evaluate the vulnerability of the full processor for a given application workload.

3.3.1 Full-Processor Vulnerability Factor

For evaluating the vulnerability of the full processor for a given application workload, we propose to extend the AVF to estimate what we refer to as the **Full-Processor Vulnerability Factor (FPVF)**. It is defined as the ratio of the total number of vulnerable bits (*VulnerableBits*) in the processor pipeline for the duration they are vulnerable (*VulnerableTime*) to the total number of bits in the processor

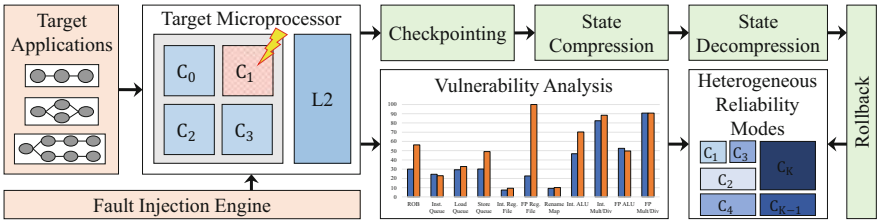


Fig. 11 Methodology for hardening out-of-order superscalar processors (adapted from [33])

pipeline (*TotalBits*) for the total duration of application execution (*TotalTime*). For a given application workload (W), we estimate FPVF of our proposed reliability-heterogeneous processors as:

$$FPVF_W = \frac{\sum_{\forall i \in Components} VulnerableBits_i \times VulnerableTime_i}{\sum_{\forall i \in Components} TotalBits_i \times TotalTime_i} \tag{3}$$

3.3.2 Heterogeneous Reliability Modes for ALPHA Cores

Enabling full-scale TMR for all application workloads leads to 200% (or more) area and power overheads, which might not be a feasible option in many real-world systems. Considering the analysis presented in Sect. 3.2, which illustrates that the AVF of the pipeline components varies based on the application workload, *we propose to enable fine-grained TMR at the component-level*. This involves hardening a combination of highly vulnerable pipeline components, instead of the full-processor pipeline to increase processor reliability while reducing the power and area overheads associated with TMR. Hardening involves instantiating three instances of the component with the same set of inputs and a voter circuit that is used to elect the majority output. We propose and analyze 10 different reliability modes (RM) for heterogeneous processors, including the baseline unprotected (U) core. The list of components hardened in these modes are presented in Table 1.

Next, we execute the four *MiBench* application benchmarks on our 10 proposed RMs to estimate the FPVF of each reliability-heterogeneous processor. We also evaluate the area and power overheads incurred by each reliability mode. The results of the experiments are illustrated in Fig. 12. From these results, we make the following **key observations:**

1. Our initial hypothesis, which stated that hardening different combinations of pipeline components (RMs) can reduce the vulnerability to different extents based on the application workload being executed, was correct. We demonstrate this further by considering the applications *SHA* and *Dijkstra*. Typically, the vulnerability of these two applications is similar to each other, except in the cases

Table 1 Heterogeneous reliability modes and corresponding pareto-optimal reliability modes for *MiBench* applications

Reliability mode	Components hardened	Application	Pareto-optimal reliability modes
U	Unprotected	Bit-counts	U, RM4, RM7
RM1	RF	Dijkstra	U, RM4, RM7, RM8
RM2	IQ, RM	Patricia	U, RM4, RM7
RM3	IQ, LQ, SQ	SHA	U, RM1, RM6, RM7, RM8
RM4	IQ, LQ, SQ, RM, ROB	All	U, RM4, RM7, RM8
RM5	RF, IQ, LQ, SQ		
RM6	RF, RM		
RM7	RF, RM, ROB		
RM8	RM, ROB		
RM9	RF, IQ, LQ, SQ, RM		

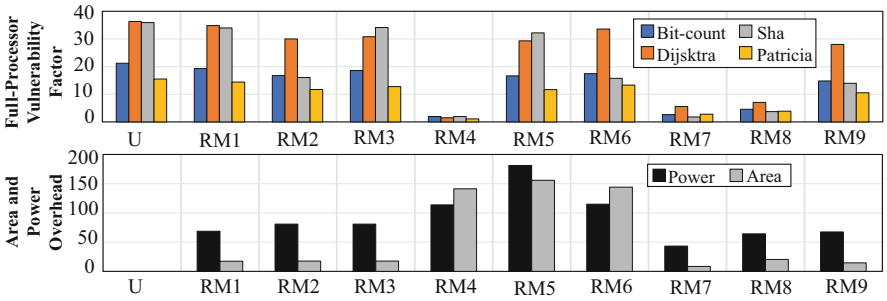


Fig. 12 Full-Processor Vulnerability Factor (FPVF) and power/area trade-off of the proposed heterogeneous reliability modes for different *MiBench* applications (adapted from [33])

of RM2, RM6, and RM9. The full-processor vulnerability of these three RMs has been reduced by more than 50% when executing *SHA* compared to *Dijkstra*.

2. Components such as the Rename Map and Reorder Buffer, when hardened, are highly effective in reducing the FPVF for all four applications. This is illustrated by the reliability modes RM4, RM7, and RM8, which have significantly lower FPVFs compared to their counter-parts. However, these two components occupy a significant percentage of the on-chip resources and hardening them leads to significant area and power overheads as illustrated by Fig. 12. This leads us to infer that hardening specific highly vulnerable pipeline components can significantly reduce the overall processor vulnerability for a wide range of application workloads based on their properties.

Furthermore, based on the data from these experiments, we perform an architectural space exploration that trades-off FPVF, area, and power overheads to extract the pareto-optimal reliability modes. The results of the experiments are illustrated in Fig. 13, where the *x*-, *y*-, and *z*-axes depict the FPVF, area, and power overheads, respectively. From these results, we make the following **key observations:**

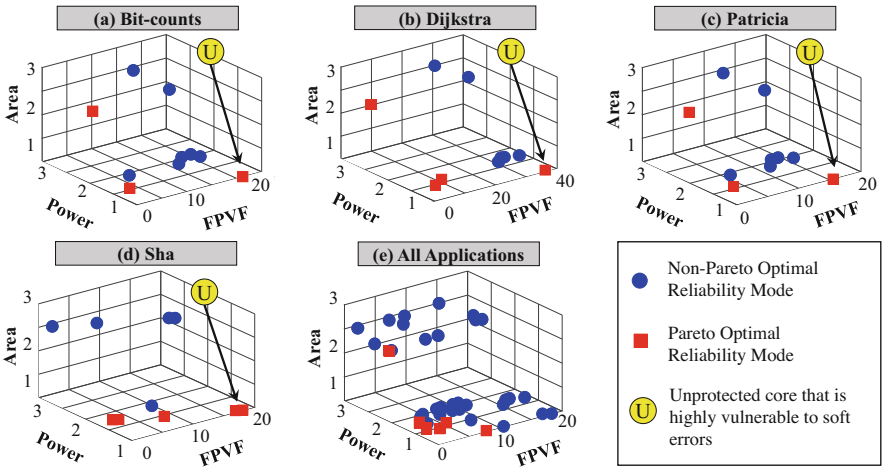


Fig. 13 Architectural space exploration of our heterogeneous reliability modes for *MiBench* applications (adapted from [33])

1. The design labeled *U*, i.e., the unprotected core, is pareto-optimal for all application workloads. This is expected as this reliability mode incurs zero area and power overheads and represents the least reliable processor design.
2. Although RM7 and RM8 significantly reduce the FPVF, due to their differences in power and area overheads, RM7 lies on the pareto-front for all individual application workloads, whereas RM8 is pareto-optimal only for *SHA* and *Dijkstra*. Similarly, RM4 is pareto-optimal for three of the four application workloads.
3. RM4, RM7, and RM8, all lie on the pareto-front when all applications are executed on the cores. This behavior is observed because of the varying levels of vulnerability savings achieved by the RMs when compared to their area and power overheads.
4. RM7 is pareto-optimal for four individual application workloads and reduces the FPVF by 87%, on average, while incurring area and power overheads of 10% and 43%, respectively.

3.3.3 State Compression Techniques

Reliability can also be improved at the software layer by inserting checkpoints in the application code. When an application encounters a checkpoint, the complete processor state, including all intermediate register and cache values, is stored in the main memory. These checkpoint states can be used to re-initialize the processor, which is referred to as rollback, in case a failure is detected and the next sequence of instructions are re-executed.

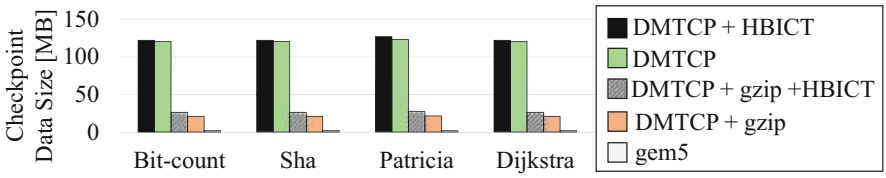


Fig. 14 Effectiveness of state compression techniques in reducing state size (adapted from [33])

The way checkpointing is implemented in *gem5* leads to significant loss in performance in case of frequent checkpoint restoration as the cache and pipeline states are not preserved, which, in turn, leads to a higher number of instructions being executed. *Distributed Multi-Threaded Checkpointing (DMTCP)* is a Linux compatible checkpointing tool that is used to checkpoint Linux processes. The back-end mechanism of DMTCP is accessible to programmers, via Application Programming Interfaces (APIs), to insert checkpoints into their application code. Inside *gem5*, these APIs can be used in combination with its pseudo-instructions to offer the functionality of creating/recovering checkpoint states for the applications being simulated inside *gem5*. Furthermore, the size of data generated by each checkpoint is typically large, especially in the case of O3 superscalar processors with large multi-level cache hierarchies. Therefore, we explore various compression strategies that can be used to efficiently compress and reduce the checkpoint data using techniques like the *Hash-Based Incremental Checkpointing Tool (HBICT)* and *GNU zip (gzip)*. HBICT provides DMTCP support to enable checkpoint compression using an approach called *delta compression*. This kind of compression mechanism preserves only changed fragments of a program's state, thereby considerably reducing the size of checkpoint data. gzip is a file compression technique based on the *DEFLATE algorithm*, which is a combination of lossless data compression techniques such as *LZ77* and *Huffman coding*. gzip can drastically reduce the size of checkpoint data, as illustrated by the results presented in Fig. 14. These techniques and compression algorithms are implemented in *gem5*, in different combinations, to reduce the size of checkpoint data for the four aforementioned *MiBench* applications, by executing them on an unprotected *ALPHA* processor. The effectiveness of different combinations of compression algorithms is illustrated in terms of checkpoint data size in Fig. 14. It can be observed that the combination of DMTCP and gzip is highly successful in reducing the checkpoint size by $\sim 6\times$. On the other hand, a combination of DMTCP, HBICT, and gzip techniques reduces the checkpoint size by $\sim 5.7\times$.

4 Run-Time Systems for Heterogeneous Fault-Tolerance

The techniques discussed in Sects. 2 and 3 also require a run-time manager for incorporating the application vulnerabilities with respect to several reliability threats

such as soft errors, aging, and process variation, as well as considering constraints like dark silicon and required performance (or tolerable performance overhead). Most of the adaptive hardware techniques exploit the application vulnerability to map applications on appropriate cores to reduce their vulnerability. Similarly, this concept can be applied to modify the applications with respect to the available hardened core or caches, which can also be combined with other hardware techniques to further reduce the vulnerabilities of the heterogeneous multi/many-core processors. Therefore, several techniques have been proposed to modify the execution patterns of the application or partitioning the application to develop a run-time system for reliability-heterogeneous multi/many-core processors.

1. **Aging- and Process Variation-Aware Redundant Multithreading [18, 36]:** *dTune* leverages multiple reliable versions of an application and redundant multithreading (RMT) simultaneously for achieving high soft error resilience under aging and process variability [36]. Based on the reliability requirements of the executing applications, *dTune* performs efficient core allocation for RMT while considering the aging state of the processor as well as process variation. It achieves up to 63% improvement in the reliability of a given application. Similarly, another approach [18] utilizes different software versions and RMT to improve the reliability of a system while considering the effects of soft errors and aging on the processor cores, to achieve an improved aging balancing.
2. **Variability-aware reliability-heterogeneous processor [21]:** This work leverages techniques at the hardware and run-time system layers to mitigate the reliability threats. In particular, this work focuses on TMR-based solutions to (partially) harden the cores for developing a many/multi-core reliability-heterogeneous processor. It uses a run-time controller to handle multiple cores with different reliability modes while considering the reliability requirements of the applications. In addition, it also exploits the dark silicon property in multi/many-core processors to offer a wide range of different performance-reliability trade-offs by over-provisioning the processor with reliability-heterogeneous cores.
3. **Aging-aware reliability-heterogeneous processor [10]:** This technique exploits the dark silicon property of the multi/many-core processors to design a run-time approach for balancing the application load to mitigate the reliability threats, i.e., temperature-dependent aging while also considering variability and current age of the cores in order to improve the overall system performance for a given lifetime constraint. The analysis shows that this run-time solution can improve the overall aging of the multi/many-core processor by 6 months to 5 years depending upon the provided design constraints and power overheads. Furthermore, this work also developed a fast aging evaluation methodology based on multi-granularity simulation epochs, as well as lightweight run-time techniques for temperature and aging estimation that can be used for an early estimation of temperature-dependent aging of multi/many-core processors.

There are other techniques which can exploit the functional and timing reliability in real-time systems to improve the application by generating the reliable application

versions or respective thread with different performance and reliability properties [38]. These reliable applications or respective thread can jointly be used with hardware techniques to improve the overall reliability of the multi/many-core heterogeneous processor. Another solution is to exploit the dynamic voltage and frequency scaling to generate the dynamic redundancy and voltage scaling with respect to the effects of process variations, application vulnerability, performance overhead, and design constraints [40]. This technique demonstrates up to 60% power reductions while improving the reliability significantly. Similarly, in addition to redundancy, multiple voltage-frequency levels are introduced while considering the effects of dark silicon in multi/many-core heterogeneous processor [39]. This technique also considers the effects of soft errors and process variations in their reliability management system that provides up to 19% improved reliability under different design constraints [35]. Most of the abovementioned approaches are focused on general purpose microprocessors; however, in application-specific instruction set processors (ASIPs), the hardware hardening and corresponding runtime software assisted recovery techniques can be used to improve the soft error vulnerabilities in ASIP-based multi/many-core systems. For example, dynamic core adaptation and application specificity can be exploited to generate a processor configuration which performs the error (caused by soft error) recovery for a particular application under the given area, power, and performance constraints [24–26]. Moreover, the baseline instruction set of the targeted ASIPs can also be modified or extended to enable the error recovery functionality [23].

5 Conclusion

This chapter discusses the building blocks of computing systems (both embedded and superscalar processors) with different heterogeneous fault-tolerant modes for the memory components like caches as well as for the in-order and out-of-order processor designs. We provide a comprehensive vulnerability analysis of different components, i.e., embedded and superscalar, processors and caches, considering the soft errors and aging issues. We also discuss the methodologies to improve the performance and power of such systems by exploiting these vulnerabilities. In addition, we briefly present that a reliability-aware compiler can be leveraged to comprehend software-level heterogeneous fault-tolerance by generating different reliable versions of the application with respective reliability and performance properties. Further details on reliability-driven compilation can be found in Chap. 5. Towards the end, we also analyze fault-tolerance techniques for application-specific instruction set processors (ASIPs).

Acknowledgments This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500—spp1500.itec.kit.edu). We would like to thank Arun Subramaniyan, Duo Sun and Segnon Jean Bruno Ahandagbe for their contributions to parts of the works cited in this chapter.

References

1. Agarwal, S., Garg, R., Gupta, M.S., Moreira, J.E.: Adaptive incremental checkpointing for massively parallel systems. In: Proceedings of the 18th Annual International Conference on Supercomputing, pp. 277–286. ACM, New York (2004)
2. Agarwal, M., Paul, B.C., Zhang, M., Mitra, S.: Circuit failure prediction and its application to transistor aging. In: 25th IEEE VLSI Test Symposium (VTS'07), pp. 277–286. IEEE, Piscataway (2007)
3. Ansel, J., Arya, K., Cooperman, G.: DMTCP: transparent checkpointing for cluster computations and the desktop. In: 2009 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–12. IEEE, Piscataway (2009)
4. Baumann, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans. Device Mater. Reliab. **5**(3), 305–316 (2005)
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH Comput. Archit. News **39**(2), 1–7 (2011)
6. Chen, C.L., Hsiao, M.: Error-correcting codes for semiconductor memory applications: a state-of-the-art review. IBM J. Res. Dev. **28**(2), 124–134 (1984)
7. Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: 2011 38th Annual International Symposium on Computer Architecture (ISCA), pp. 365–376. IEEE, Piscataway (2011)
8. Gailly, J.: Gzip—the data compression program (1993)
9. Geist, A.: Supercomputing's monster in the closet. IEEE Spectr. **53**(3), 30–35 (2016)
10. Gnad, D., Shafique, M., Kriebel, F., Rehman, S., Sun, D., Henkel, J.: Hayat: harnessing dark silicon and variability for aging deceleration and balancing. In: Proceedings of the 52nd Annual Design Automation Conference, San Francisco, June 7–11, pp. 180:1–180:6 (2015)
11. 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 Integr. Circuits Syst. **32**(1), 8–23 (2013). https://doi.org/10.1109/TCAD.2012.2223467
12. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538), pp. 3–14. IEEE, Piscataway (2001)
13. 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, 9–14 October, 2011, pp. 69–78 (2011). https://doi.org/10.1145/2039370.2039384
14. Henkel, J., Bauer, L., Dutt, N., Gupta, P., Nassif, S., Shafique, M., Tahoori, M., Wehn, N.: Reliable on-chip systems in the nano-era: lessons learnt and future trends. In: Proceedings of the 50th Annual Design Automation Conference, p. 99. ACM, New York (2013)
15. Jeyapaul, R., Shrivastava, A.: Enabling energy efficient reliability in embedded systems through smart cache cleaning. ACM Trans. Des. Autom. Electron. Syst. **18**(4), 53 (2013)
16. Kang, K., Gangwal, S., Park, S.P., Roy, K.: NBTI induced performance degradation in logic and memory circuits: How effectively can we approach a reliability solution? In: Proceedings of the 2008 Asia and South Pacific Design Automation Conference, pp. 726–731. IEEE Computer Society Press, Silver Spring (2008)
17. Kessler, R.E.: The alpha 21264 microprocessor. IEEE Micro **19**(2), 24–36 (1999)
18. Kriebel, F., Rehman, S., Shafique, M., Henkel, J.: ageOpt-RMT: compiler-driven variation-aware aging optimization for redundant multithreading. In: Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, June 5–9, 2016, pp. 46:1–46:6 (2016). https://doi.org/10.1145/2897937.2897980

19. Kriebel, F., Rehman, S., Subramaniyan, A., Ahandagbe, S.J.B., Shafique, M., Henkel, J.: Reliability-aware adaptations for shared last-level caches in multi-cores. ACM Trans. Embed. Comput. Syst. **15**(4), 67:1–67:26 (2016). https://doi.org/10.1145/2961059
20. Kriebel, F., Rehman, S., Sun, D., Shafique, M., Henkel, J.: ASER: Adaptive soft error resilience for reliability-heterogeneous processors in the dark silicon era. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM, New York (2014)
21. Kriebel, F., Shafique, M., Rehman, S., Henkel, J., Garg, S.: Variability and reliability awareness in the age of dark silicon. IEEE Des. Test **33**(2), 59–67 (2016)
22. Kriebel, F., Subramaniyan, A., Rehman, S., Ahandagbe, S.J.B., Shafique, M., Henkel, J.: R^2cache: reliability-aware reconfigurable last-level cache architecture for multi-cores. In: 2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, October 4–9, 2015, pp. 1–10 (2015)
23. Li, T., Shafique, M., Ambrose, J.A., Henkel, J., Parameswaran, S.: Fine-grained checkpoint recovery for application-specific instruction-set processors. IEEE Trans. Comput. **66**(4), 647–660 (2017)
24. Li, T., Shafique, M., Ambrose, J.A., Rehman, S., Henkel, J., Parameswaran, S.: RASTER: runtime adaptive spatial/temporal error resiliency for embedded processors. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–7. IEEE, Piscataway (2013)
25. Li, T., Shafique, M., Rehman, S., Ambrose, J.A., Henkel, J., Parameswaran, S.: DHASER: dynamic heterogeneous adaptation for soft-error resiliency in ASIP-based multi-core systems. In: 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 646–653. IEEE, Piscataway (2013)
26. Li, T., Shafique, M., Rehman, S., Radhakrishnan, S., Ragel, R., Ambrose, J.A., Henkel, J., Parameswaran, S.: CSER: HW/SW configurable soft-error resiliency for application specific instruction-set processors. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 707–712. EDA Consortium, San Jose (2013)
27. McPherson, J.W.: Reliability challenges for 45 nm and beyond. In: 2006 43rd ACM/IEEE Design Automation Conference, pp. 176–181. IEEE, Piscataway (2006)
28. Mitra, S., Seifert, N., Zhang, M., Shi, Q., Kim, K.S.: Robust system design with built-in soft-error resilience. Computer **38**(2), 43–52 (2005)
29. Mukherjee, S.S., Emer, J., Reinhardt, S.K.: The soft error problem: an architectural perspective. In: 11th International Symposium on High-Performance Computer Architecture, pp. 243–247. IEEE, Piscataway (2005)
30. Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36, pp. 29–40. IEEE, Piscataway (2003)
31. Pagani, S., Khdr, H., Munawar, W., Chen, J.J., Shafique, M., Li, M., Henkel, J.: TSP: thermal safe power: efficient power budgeting for many-core systems in dark silicon. In: Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis, p. 10. ACM, New York (2014)
32. Portolan, M., Leveugle, R.: A highly flexible hardened RTL processor core based on LEON2. IEEE Trans. Nucl. Sci. **53**(4), 2069–2075 (2006)
33. Prabakaran, B.S., Dave, M., Kriebel, F., Rehman, S., Shafique, M.: Architectural-space exploration of heterogeneous reliability and checkpointing modes for out-of-order superscalar processors. IEEE Access **7**, 145324–145339 (2019)
34. Rehman, S., Kriebel, F., Prabakaran, B.S., Khalid, F., Shafique, M.: Hardware and software techniques for heterogeneous fault-tolerance. In: 24th IEEE International Symposium on On-Line Testing And Robust System Design, IOLTS 2018, Platja D'Aro, July 2–4, 2018, pp. 115–118 (2018). https://doi.org/10.1109/IOLTS.2018.8474219
35. Rehman, S., Kriebel, F., Shafique, M., Henkel, J.: Reliability-driven software transformations for unreliable hardware. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(11), 1597–1610 (2014)

36. Rehman, S., Kriebel, F., Sun, D., Shafique, M., Henkel, J.: dTune: leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM, New York (2014)
37. Rehman, S., Shafique, M., Henkel, J.: Reliable Software for Unreliable Hardware: A Cross Layer Perspective. Springer, Berlin (2016)
38. Rehman, S., Toma, A., Kriebel, F., Shafique, M., Chen, J.J., Henkel, J.: Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 273–282. IEEE, Piscataway (2013)
39. Salehi, M., Shafique, M., Kriebel, F., Rehman, S., Tavana, M.K., Ejlali, A., Henkel, J.: dsReliM: power-constrained reliability management in Dark-Silicon many-core chips under process variations. In: 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 75–82. IEEE, Piscataway (2015)
40. Salehi, M., Tavana, M.K., Rehman, S., Kriebel, F., Shafique, M., Ejlali, A., Henkel, J.: DRVS: power-efficient reliability management through dynamic redundancy and voltage scaling under variations. In: 2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), pp. 225–230. IEEE, Piscataway (2015)
41. Shafique, M., Garg, S., Henkel, J., Marculescu, D.: The EDA challenges in the dark silicon era: temperature, reliability, and variability perspectives. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM, New York (2014)
42. Shafique, M., Rehman, S., Aceituno, P.V., Henkel, J.: Exploiting program-level masking and error propagation for constrained reliability optimization. In: Proceedings of the 50th Annual Design Automation Conference, p. 17. ACM, New York (2013)
43. Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings International Conference on Dependable Systems and Networks, pp. 389–398. IEEE, Piscataway (2002)
44. Srinivasan, J., Adve, S.V., Bose, P., Rivers, J.A.: The case for lifetime reliability-aware microprocessors. In: ACM SIGARCH Computer Architecture News, vol. 32, p. 276. IEEE Computer Society, Silver Spring (2004)
45. Subramaniyan, A., Rehman, S., Shafique, M., Kumar, A., Henkel, J.: Soft error-aware architectural exploration for designing reliability adaptive cache hierarchies in multi-cores. In: Design, Automation and Test in Europe Conference and Exhibition, DATE 2017, Lausanne, March 27–31, 2017, pp. 37–42 (2017)
46. Vadlamani, R., Zhao, J., Burleson, W., Tessier, R.: Multicore soft error rate stabilization using adaptive dual modular redundancy. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 27–32. European Design and Automation Association (2010)
47. Wang, W., Mishra, P.: Dynamic reconfiguration of two-level cache hierarchy in real-time embedded systems. J. Low Power Electron. **7**(1), 17–28 (2011)

Thermal Management and Communication Virtualization for Reliability Optimization in MPSoCs

Victor M. van Santen, Hussam Amrouch, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf

1 Overview

The VirTherm3D project is part of SPP1500, which has its origins in [10] and [9]. The main cross-layer contributions of VirTherm3D are outlined in Fig. 1. The green circles are our major contributions spanning from the physics to circuit layer and from the architecture to application layer. These contributions include physical modeling of thermal and aging effects considered at the circuit layer as well as communication virtualization at architecture level to support task relocation as part of thermal management at architecture level. Our minor contributions span from the circuit to architecture layer and include reliability-aware logic synthesis as well as studying the impact of reliability with figures of merit such as probability of failure.

2 Impact of Temperature on Reliability

Temperature is at the core of reliability. It has a direct short-term impact on reliability, as the electrical properties of circuits (e.g., delay) are affected by temperature. A higher temperature leads to circuits with higher delays and lower noise margins. Additionally, temperature impacts circuits indirectly as it stimulates or accelerates aging phenomena, which in turn, manifest themselves as degradations in the electrical properties of circuits.

The direct impact of temperature in an SRAM memory cell can be seen in Fig. 2. Increasing the temperature increases the read delay of the memory cell.

V.M. van Santen · H. Amrouch (✉) · T. Wild · J. Henkel · A. Herkersdorf
Karlsruhe Institute of Technology, Technical University of Munich, Munich, Germany
e-mail: victor.santen@kit.edu; amrouch@kit.edu; thomas.wild@tum.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_8

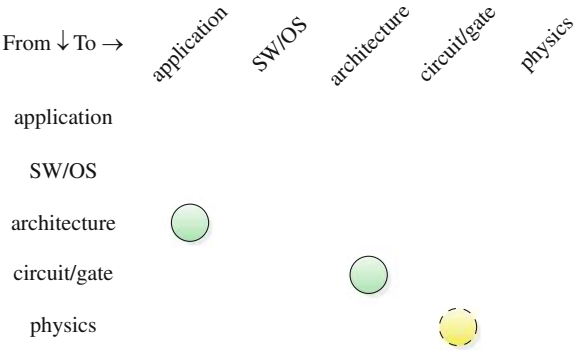


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

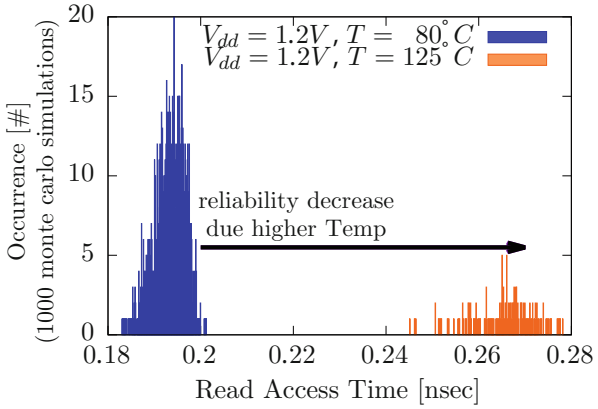


Fig. 2 Shift in SRAM memory cell read delay as a direct impact of temperature. Taken (from [3])

This is because increased temperature degrades performance of transistors (e.g., a reduction in carrier mobility μ), which affects the performance of the memory cell. Therefore, increasing temperature directly worsens circuit performance and thus negatively impacts the reliability of a circuit. If the circuit has a prolonged delay due to the increased temperature, then timing violations might occur. If the circuit has a degraded noise margin, then noise (e.g., voltage drops or radiation-induced current spikes) might corrupt data.

Next to directly altering the circuit properties, temperature also has an indirect impact, which is shown in Fig. 3. Temperature stimulates aging phenomena (e.g., Bias Temperature Instability (BTI)) degrading the performance of transistors (e.g., increasing the threshold voltage V_{th}) over time. Increasing the temperature accelerates the underlying physical processes of aging and thus increases aging-induced degradations.

Because of the two-fold impact of temperature, i.e., by reducing circuit performance directly and indirectly via aging, it is crucial to be considered when

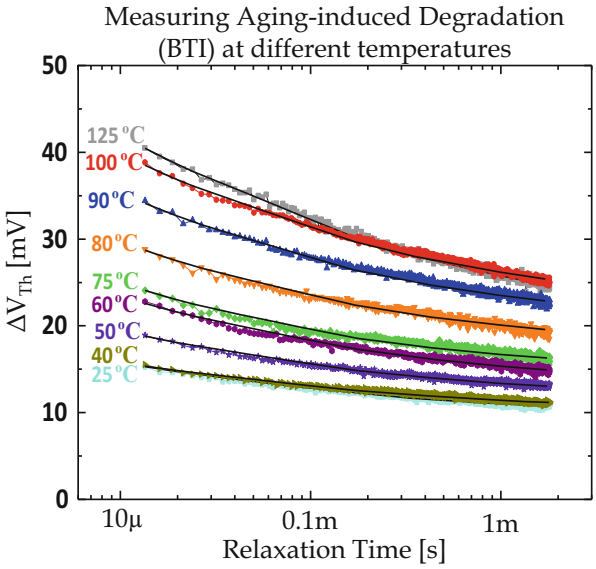


Fig. 3 Indirect impact of temperature stimulating aging. Taken (from [4])

estimating the reliability of a circuit. The temperature at an instant of time (estimated either via measurement or simulation) governs the direct degradation of the circuit, i.e., the short-term direct impact of temperature. Temperature over time governs the long-term indirect impact, as aging depends on the thermal profile (i.e., the thermal fluctuations over a long period). How to estimate temperature correctly both the temperature at an instant as well as the thermal profile is discussed in Sect. 3.

After the temperature is determined via temperature estimation, the impact of temperature on reliability must be evaluated. This is challenging, as the impact of temperature occurs on physical level (e.g., movement of electrical carriers in a semiconductor as well as defects in transistors for aging), while the figures of merit are for entire computing systems (e.g., probability of failure, quality of service). To overcome this challenge, Sect. 4 discusses how to connect the physical to the system level with respect to thermal modeling. To obtain the ultimate impact of the temperature, the figures of merit of a computing system are obtained with our cross-layer (from physical to system level) temperature modeling (see Fig. 4).

Temperature can be controlled. Thermal management techniques reduce temperature by limiting the amount of generated heat or making better use of existing cooling (e.g., distribution of generated heat for easier cooling). Thus, to reduce the deleterious impact of temperature on the figures of merit of systems, temperature must be controlled at system level. For this purpose, Sect. 5 discusses system-level thermal management techniques. These techniques limit temperature below a specified critical temperature to ensure that employed safety margins (e.g., are not violated time slack to tolerate thermally induced delay increases), thus ensuring the reliability of a computing system.

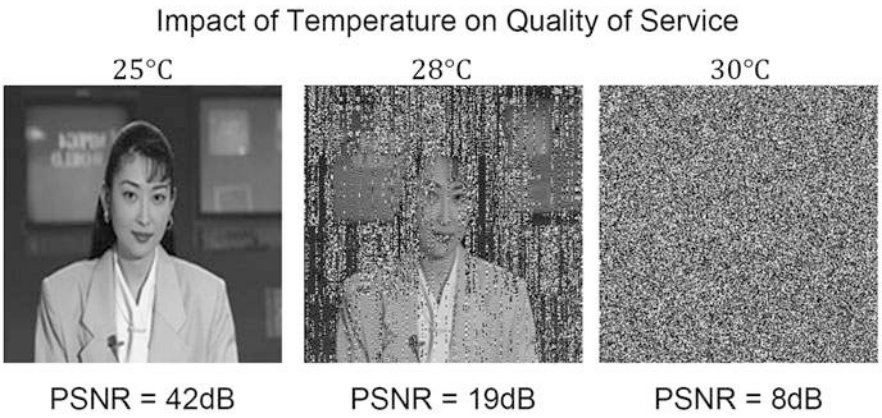


Fig. 4 Image signal-to-noise ratio as a figure of merit for an image processing system. At 25 °C no timing violations occur, while (due to non-existent safety margins) at 28 and 30 °C timing violations degrade PSNR

To support the system management in migrating tasks away from thermal hotspots and thus reducing thermal stress, special virtualization features are proposed to be implemented in the interconnect infrastructure. They allow for a fast transfer of communication relations of tasks to be migrated and thus help to limit downtimes. These mechanisms can then also be applied for generating task replica to dynamically introduce redundancy during system runtime as a response to imminent reliability concerns in parts of the SoC or if reliability requirements of an application change.

As mentioned before, temperature estimation and modeling cross many abstraction layers. The effects of temperature originate from the physical level, where the physical processes related to carriers and defects are altered by temperature. Yet the final impact of temperature has to pass through the transistor level, gate level, circuit level, architecture level all the way to the system level, where the figures of merit of the system can be evaluated. The system designer has to maintain the figures of merit for his end-user, therefore limiting temperature with thermal management techniques and evaluating the impact of temperature on the various abstraction layers. Therefore, Sect. 7 discusses thermal estimation, modeling, and management techniques with a focus on how to cross these abstraction layers and how to connect the physical to the system level. In practice, interdependencies between the low abstraction layers and the management layer do exist. The running workload at the system level increases the temperature of the cores. Hence, the probability of error starts to gradually increase. In such a case the management layer estimates the probability of error based on the information received from the lower layers and then attempts to make the best decision. For instance, it might allow the increase in the probability of error but at the cost of enabling the adaptive modular redundancy (AMR) (details in Sect. 6.3) or maybe migrating the tasks to other cores that are healthier (i.e., exhibit less probability of error).

3 Temperature Estimation via Simulation or Measurement

Accurately estimating the temperature of a computing system is necessary to later evaluate the impact of temperature. Two options exist: (1) Thermal simulation and (2) Thermal measurement. Both options must estimate temperature with respect to time and space. Figure 5 shows a simulated temporal thermal profile of a microprocessor. Temperature fluctuates visibly over time and depends on the applications which are run on the microprocessor.

Figure 9 shows a measured spatial thermal map of a microprocessor. Temperature is spatially unequally distributed across the processor, i.e., certain components of the microprocessors have to tolerate higher temperatures. However, the difference in temperature is limited. This limit stems from thermal conductance across the chip counteracting temperature differences. Thermal conductance is mainly via the chip itself (e.g., wires in metal layers), its packaging (e.g., heat spreaders), and cooling (e.g., heat sink).

3.1 Thermal Simulation

Thermal simulations are a software-based approach to estimate the temperature of a computing system. Thermal simulations consist of three steps: (1) Activity extraction, (2) Power estimation, (3) Temperature estimation. The first step extracts the activity (e.g., transistor switching frequency, cache accesses) of the applications running on the computing system. Different applications result in different temperatures (see Fig. 5). The underlying cause is a unique power profile for each application (and its input data), originating from unique activities per application.

Once activities are extracted, the power profiles based on these activities are estimated. Both steps can be performed on different abstraction layers. On the transistor level, transistor switching consumes power, while on the architecture level

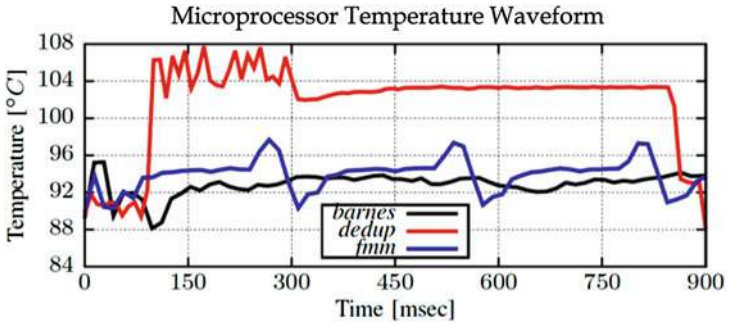


Fig. 5 Simulated thermal profile (temporal) of a microprocessor

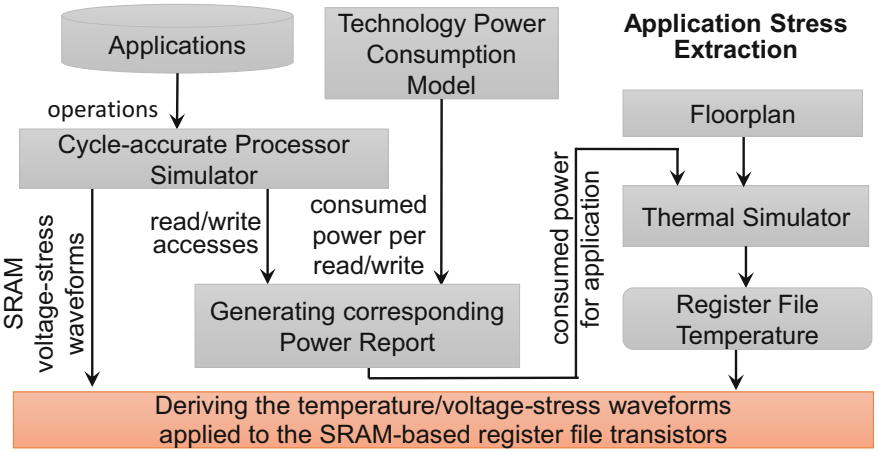


Fig. 6 Flow of a thermal simulation (updated figure from [4]).

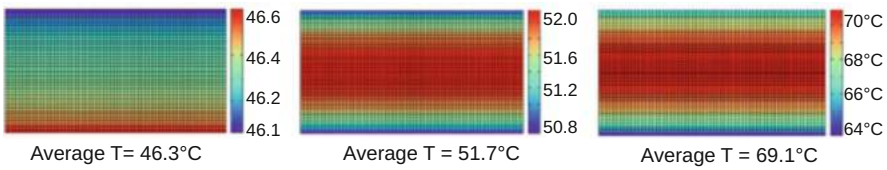


Fig. 7 Thermal map of an SRAM array (granularity: single SRAM cells) under different applications

each cache access consumes a certain amount of power (depending on cache hit or cache miss). Thus activity would be transistor switching/cache accesses and this would result in a very fine-grained power profile (temporally as well as spatially) for the transistor level. At the architecture level, a coarse-grained power profile is obtained with a time granularity per access (potentially hundreds of cycles long) and space granularity is per entire cache block.

With the power profiles known, the amount of generated heat (again spatially and temporally) is known. A thermal simulator then uses a representation of thermal conductances and capacitances with generated heat as an input heat flux and dissipated heat (via cooling) as an output heat flux to determine the temperature over time and across the circuit.

Our work in [4] exemplifies a thermal simulation flow in Fig. 6. In this example, SRAM memory cell accesses are used to estimate transistor switching and thus power profiles for the entire SRAM array. These power profiles are then used with the microprocessor layout (called floorplan) and typical cooling settings in a thermal simulator to get thermal maps in Fig. 7.

The work in [13] models temperature on the system level. Individual processor cores of a many-core computing system are the spatial granularity with seconds as

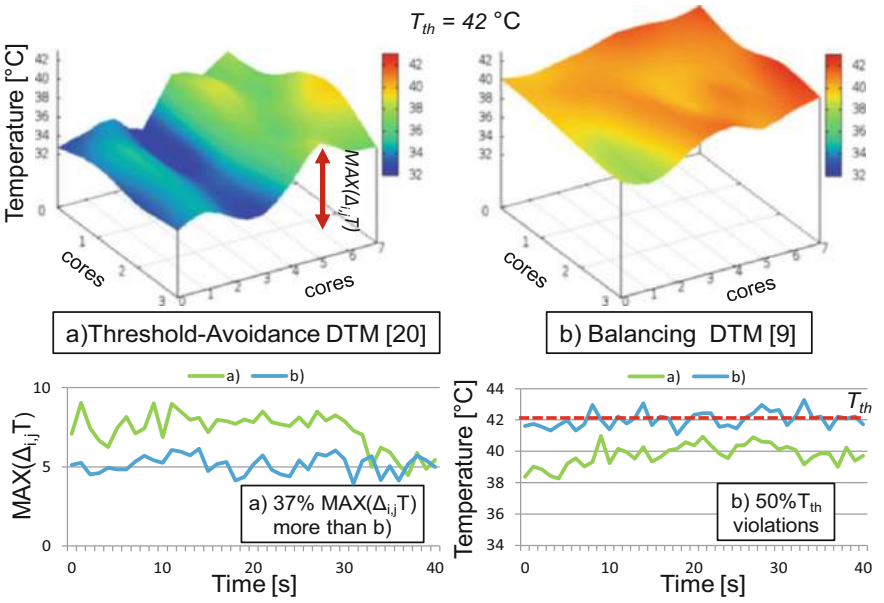


Fig. 8 Thermal estimation at the system level with cores as the spatial granularity (from [13])

the temporal granularity. Abstracted (faster, simpler) models are used to estimate the temperature per processor core, as a transistor level granularity would be unfeasible with respect to computational effort (i.e., simulation time).

While thermal simulations have the advantage of being able to perform thermal estimations without physical access to the system (e.g., during early design phases), they are very slow (hours of simulation per second of operation) and not accurate. Estimating activities and power on fine-grained granularities is an almost impossible task (layout-dependent parasitic resistances and capacitances, billions of transistors, billions of operations per second), while coarse-grained granularities provide just rough estimates of temperature due to the disregard of non-negligible details (e.g., parasitics) at these high abstraction levels (Fig. 8).

3.2 Thermal Measurement

If physical access to actual chips is an option, then thermal measurement is preferable. Observing the actual thermal profiles (temporally) and thermal maps (spatially) intrinsically includes all details (e.g., parasitics, billions of transistors, layout). Thus, a measurement can be more accurate than a simulation. Equally as important, measurements operate in real time (i.e., a second measured is also a second operated) outperforming simulations.

The challenge of thermal measurements is the resolution. The sample frequency of the measurement setup determines the temporal resolution and this is typically

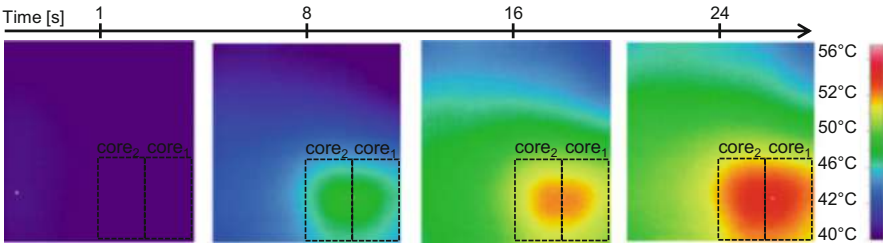


Fig. 9 Measured spatial thermal map of a microprocessor (from [2])

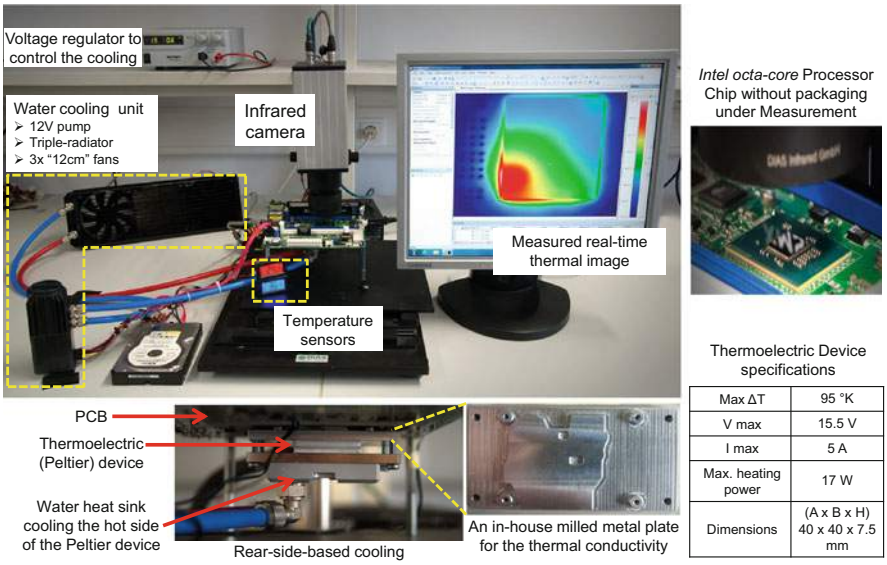


Fig. 10 High fidelity infrared thermal measurement setup (from [2])

in the order of milliseconds, while simulations can provide nano-second granularity (e.g., individual transistor switching). However, since thermal capacitances prevent abrupt changes of temperature as a reaction to abrupt changes in generated heat, sample rates in milliseconds are sufficient. The spatial resolution is equally limited by thermal conductance, which limits the thermal gradient (i.e., difference in temperature between two neighboring component; see Figs. 7 and 9).

The actual obstacle for thermal measurements is accessibility. A chip sits below a heat spreader and cooling, i.e., it is not directly observable. The manufacturers include thermal diodes at a handful of locations (e.g., 1 per core), which measure temperature in-situ, but these diodes are both inaccurate (due to their spatial separation from the actual logic) and spatially very coarse due to their limited number.

Our approach (Fig. 10) [2, 14] is to cool the chip through the PCB from the bottom-side and measure the infrared radiation emitted from the chip directly. Other

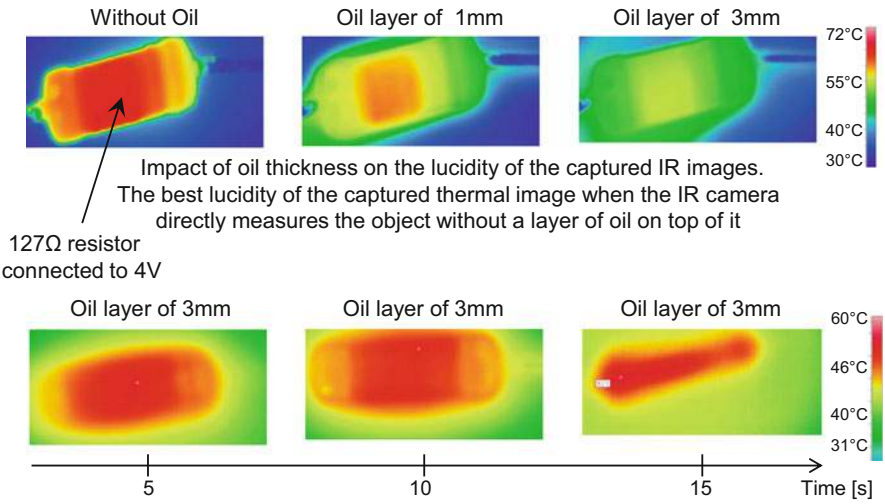


Fig. 11 Limited infrared image lucidity and fidelity due to oil cooling on top of the chip (from [2])

approaches cool the chip with infrared-transparent oils to cool the chip from the top, but this results in heat conductance limiting image fidelity and turbulence in the oil limiting image resolution (see Fig. 11). Our approach does not suffer from these issues and delivers crisp high-resolution infrared images from a camera capable of sampling an image every 20 ms with a spatial resolution of 50 µm. Thus a lucid thermal profile and thermal map are achieved including all implementation details of the chip, as actual hardware is measured.

4 Modeling Impact of Temperature at System Level

Modeling the impact of temperature on a computing system is a challenging task. Estimation of the figures of merit of a computing system can only be performed on the system level, while the effects of temperature are on the physical level. Thus, many abstraction layers have to be crossed while maintaining accuracy and computational feasibility (i.e., keep simulation times at bay). In this section we discuss how we tackle this challenge, starting with the selection of figures of merit, followed by the modeling of the direct impact of temperature and finally aging as the indirect impact of temperature.

4.1 Figures of Merit

The main figures of merit at the system level with respect to reliability are probability of failure P_{fail} and quality of service (e.g., PSNR in image processing). Probability of failure encompasses many failure types like timing violations, data corruption and catastrophic failure of a component (e.g., short-circuit). A full overview of abstraction of failures towards probability of failure is given in the RAP (Resilience Articulation Point) chapter of this book. Typically, vendors or end-users require the system designer to meet specific P_{fail} criteria (e.g., $P_{fail} < 0.01$). Quality of service describes how well a system provides its functionality if a specific amount of errors can be tolerated (e.g., if human perception is involved or for classification problems).

For probability of failure, the individual failure types have to be estimated and quantified without over-estimation due to common failures (as in our work [3], where a circuit with timing violations might also corrupt data). In that work the failure types such as timing violations, data corruption due to voltage noise, and data corruption due to strikes of high-energy particles are covered. These are the main causes of failure in digital logic circuits as a result of temperature changes (e.g., excluding mechanical stress from drops). The probability of failure is spatially and temporarily distributed (see Figs. 12 and 13) and therefore has to be estimated for a given system lifetime (temporally) and for total system failure (combined impact of spatially distributed P_{fail} (e.g., sum of failures or probability that only 1 component out of 3 fail (modular redundancy)).

Quality of service means observing the final output of the computing system and analyzing it. In our work [5, 7] we use the peak signal-to-noise ratio (PSNR) of an output image from an image processing circuit (discrete cosine transformation (DCT) in a JPEG encoder).

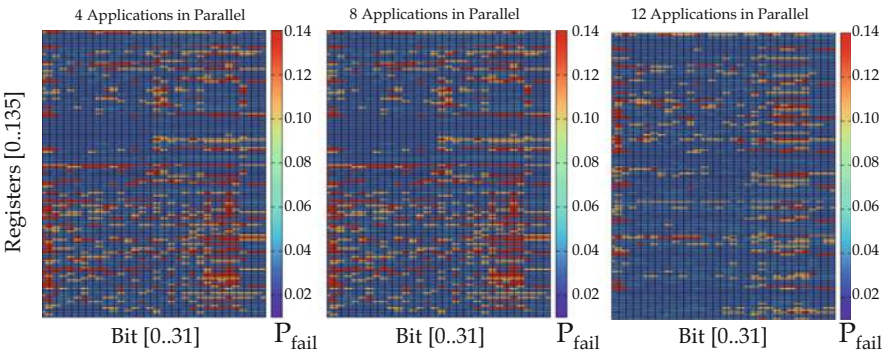


Fig. 12 Spatial distribution of P_{fail} across an SRAM array under two different applications (from [4])

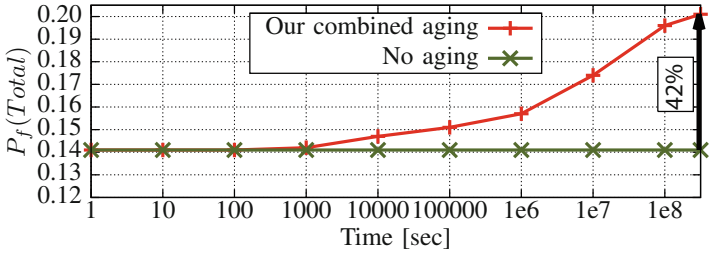


Fig. 13 Temporal distribution of P_{fail}, which increases over time due to aging (from [3])

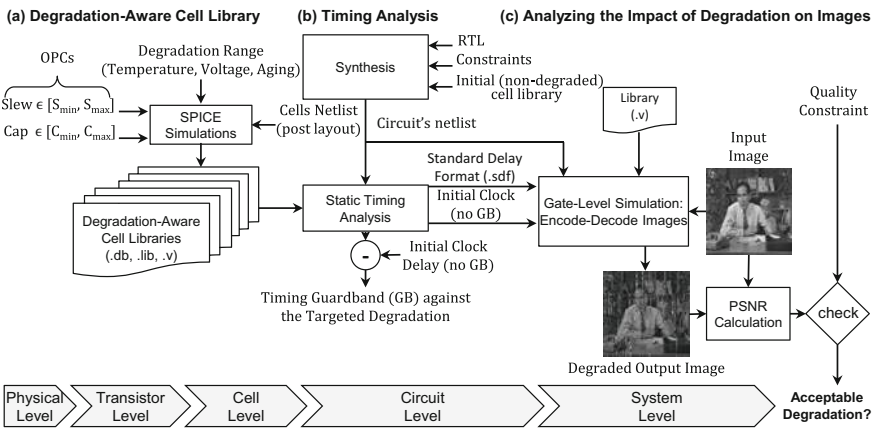


Fig. 14 Flow of the characterization of standard cells and the subsequent use of degradation-aware cell libraries to obtain timing violations (from [12])

4.2 Direct Impact of Temperature

To model the direct impact of temperature, we start at the lowest abstraction layers. Compact transistor models (e.g., BSIM) describe the current flow through the channel of a transistor and the impact of temperature on that current flow. These models are then used in circuit simulators to characterize standard cells (build from transistors) in terms of power consumption and propagation delay [5, 21]. Characterizing the standard cells (see Fig. 14) under different temperatures (e.g., from 25 to 125 °C) captures the impact of temperature on the delay and power consumption of these cells. This information is then gathered in a cell library (a single file containing all delay and power information for these cells) and then circuit and architecture level tools (e.g., static timing analysis tools, gate level simulators) can be used to check individual failure types (e.g., timing violations in static timing analysis) for computing systems (e.g., microprocessors) under various temperatures.

4.3 Aging as Indirect Impact of Temperature

Aging is stimulated by temperature (see Fig. 3) and therefore temperature has an indirect impact on reliability via aging-induced degradations. Aging lowers the resiliency of circuits and systems, thus decreasing reliability (an increase in P_{fail}) as shown in Fig. 15.

For this purpose our work [6, 18, 20] models aging, i.e., Bias Temperature Instability (BTI), Hot-Carrier Injection (HCI), Time-Dependent Dielectric Breakdown (TDDB) and the effects directly linked to aging like Random Telegraph Noise (RTN). All these phenomena are modeled with physics-based models [18, 20], which can accurately describe their temperature dependencies in the actual physical processes (typically capture and emission of carriers in the defects in the gate dielectric of transistors [4, 19]) of these phenomena.

Our work [6, 18] considers the interdependencies between these phenomena (see Fig. 16) and then estimates the degradation of the transistors. Then the transistor modelcards (transistor parameter lists) are adapted to incorporate the estimated degradations and use these degraded transistor parameters in standard cell characterization.

During cell characterization it is important to not abstract, as ignoring the interactions between transistors (counteracting each other when switching) results in underestimations of propagation delay [21] and ignoring the operating conditions

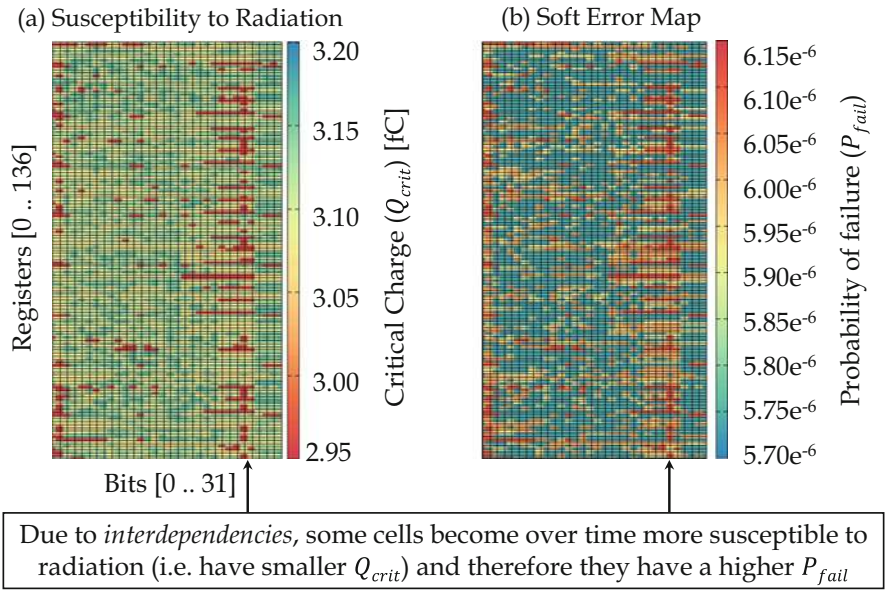


Fig. 15 Link between aging (increasing susceptibility) and P_{fail} (from [6])

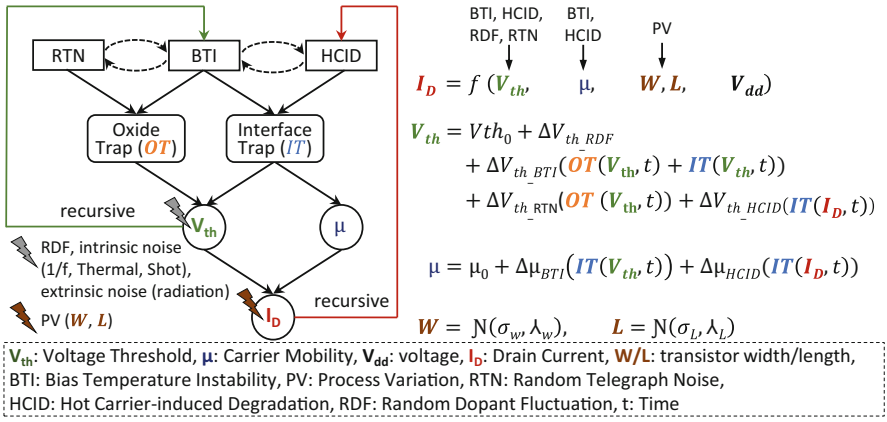


Fig. 16 Interdependencies between the aging phenomena (from [6])

(load capacitance, signal slew) of the cells [5, 20] misrepresents actual cell delay and power consumption.

After all necessary information is gathered, cells are characterized under different temperatures (like in the previous subsection) but not only with altered transistor currents (modeling the direct impact of temperature) but with additionally degraded transistors parameters (modeling the indirect impact of temperature via aging). Thus we combine both the direct and indirect impact into a single standard cell characterization to obtain delay and power information of standard cells under the joint impact of temperature and temperature-stimulated aging.

5 System-Level Management

To limit the peak temperature of a computing system and distribute the temperature evenly, we can employ system-level thermal management techniques. These techniques limit or distribute the amount of generated heat and thus ensure that the temperature stays below a given critical temperature. The two techniques presented in this section are task migration [13] and voltage scaling [17].

5.1 Voltage Scaling

Voltage scaling reduces the supply voltage of a chip or component (e.g., a processor core) to lower the power consumption and thus lower the generated heat. As a first-order approximation, lowering the voltage results in a quadratic reduction of

the consumed (dynamic) power. Therefore lowering the voltage even slightly has a considerable impact on the generated heat and thus exhibited temperature.

Voltage scaling has various side-effects. As the driving strength of transistors is also reduced, when the supply voltage is reduced, voltage scaling always prolongs circuit delays. Hence, voltage scaling has a performance overhead, which has to be minimized, while at the same time the critical temperature should not be exceeded.

Another side-effect is that voltage governs the electric field, which also stimulates aging [17]. When voltage increases, aging-induced degradation increases and when voltage reduces aging recovers (decreasing degradation). In our work in [17] we showed that voltage changes within a micro-second might induce transient timing violations. During such ultra-fast voltage changes, the low resiliency of the circuit (at the lower supply voltage) meets the high degradation of aging (exhibiting from operation at the high voltage). This combination of high degradation with low resiliency leads to timing violations if not accounted for. Continuing operation at the lower voltage recovers aging, thus resolving the issue. However, during the brief moment of high degradation violations occurred.

5.2 Task Migration

Task migration is the process of moving applications from one processor core to another. This allows a hot processing core to cool down, while a colder processor core takes over the computation of the task. Therefore, temperature is more equally distributed across a multi- or many-core computing system.

A flow of our task migration approach is shown in Fig. 17. Sensors in each core measure the current temperature (typically thermal diodes). As soon as the temperature approaches the critical value, then a task is migrated to a different core. The entire challenge is in the question "To which core is the task migrated?" If the core to which the task is migrated is only barely below the critical temperature, then the task is migrated again, which is costly since each migration stalls the processor core for many cycles (caches are filled, data has to be fetched, etc.).

Therefore our work in [13] predicts the thermal profile and makes decisions based on these predictions to optimize the task migration with as little migrations as possible while still ensuring that the critical temperature is not exceeded.

Another objective which has to be managed by our thermal management technique is to minimize thermal cycling. Each time a processor core cools down and heats up again it experiences a thermal cycle. Materials shrink and expand under temperature and thus thermal cycles put stress on bonding wires as well as soldering joints between the chip and the PCB or even interconnects within the chip (when it is partially cooled/heated).

Therefore, our approach is a multi-objective optimization strategy, which minimizes thermal cycles per core, limits temperature below the critical temperature and minimizes the number of task migrations (reducing performance overheads).

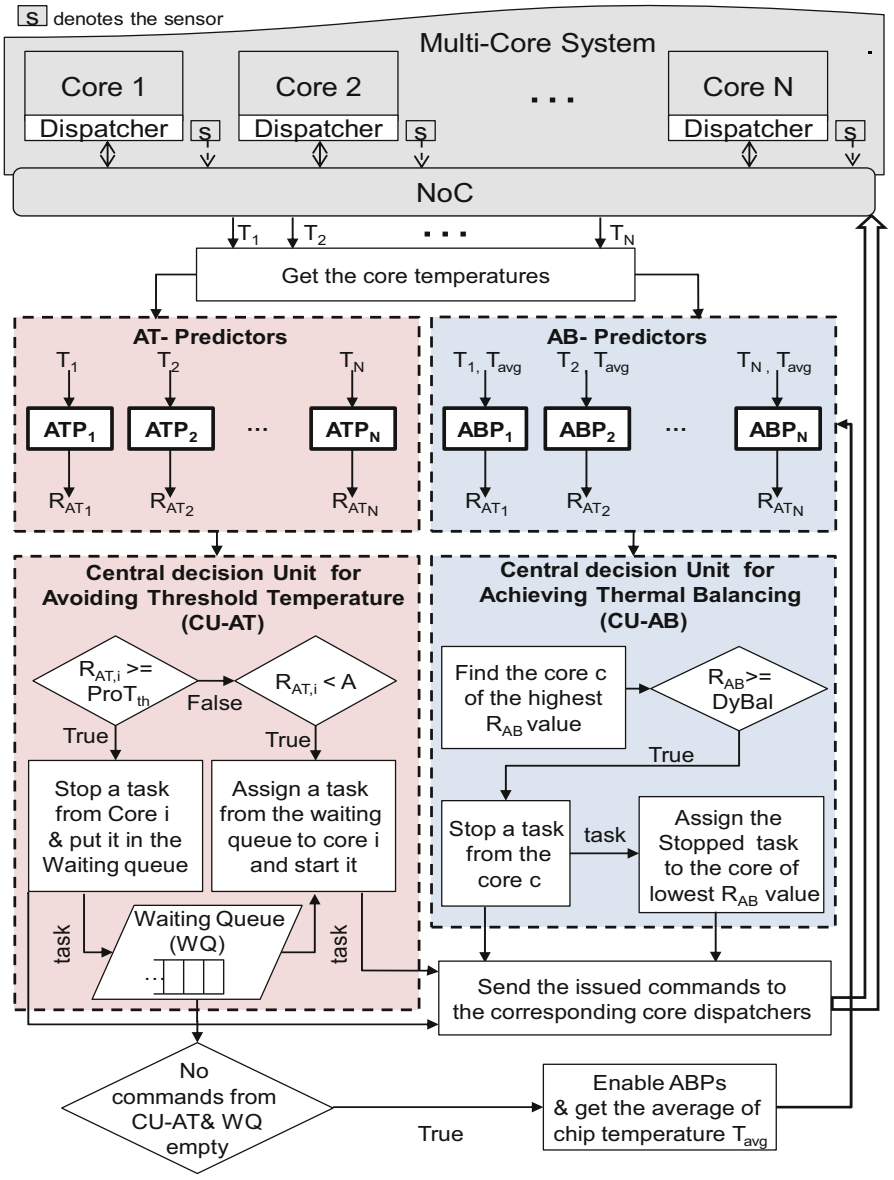


Fig. 17 Flow of our task migration approach to bound temperature in a many-core computing system (from [13])

6 Architecture Support

To support the system-level thermal management in the migration of tasks that communicate with each other the underlying hardware architecture provides specific assistance functions in the communication infrastructure. This encompasses a virtualization layer in the network on chip (NoC) and the application of protection switching mechanisms for a fast switch-over of communication channels. Based on these features an additional redundancy mechanism—called adaptive modular redundancy (AMR)—is introduced, which allows to run tasks temporarily with a second or third replica to either detect or correct errors.

6.1 NoC Virtualization

To support the system management layer in the transparent migration of tasks between processor cores within the MPSoC an interconnect virtualization overlay is introduced, which decouples physical and logical endpoints of communication channels. Any message passing communication among sub-tasks of an application or with the I/O tile is then done via logical communication endpoints. That is, a sending task transmits its data from the logical endpoint on the source side of the channel via the NoC to the logical endpoint at the destination side where the receiving task is executed. Therefore, the application only communicates on the logical layer and does not have to care about the actual physical location of sender and receiver tasks within the MPSoC. This property allows dynamic remapping of a logical to a different physical endpoint within the NoC and thus eases the transparent migration of tasks by the system management. This is shown in Fig. 18, where a communication channel from the MAC interface to task T1 can be transparently

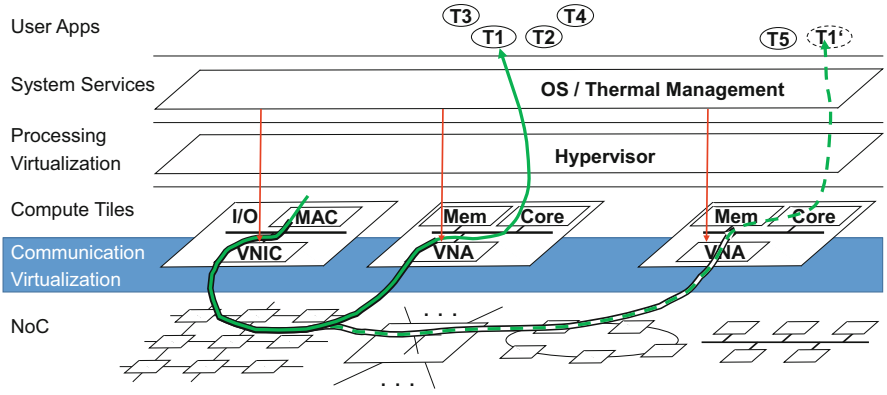


Fig. 18 Communication virtualization layer (from [8])

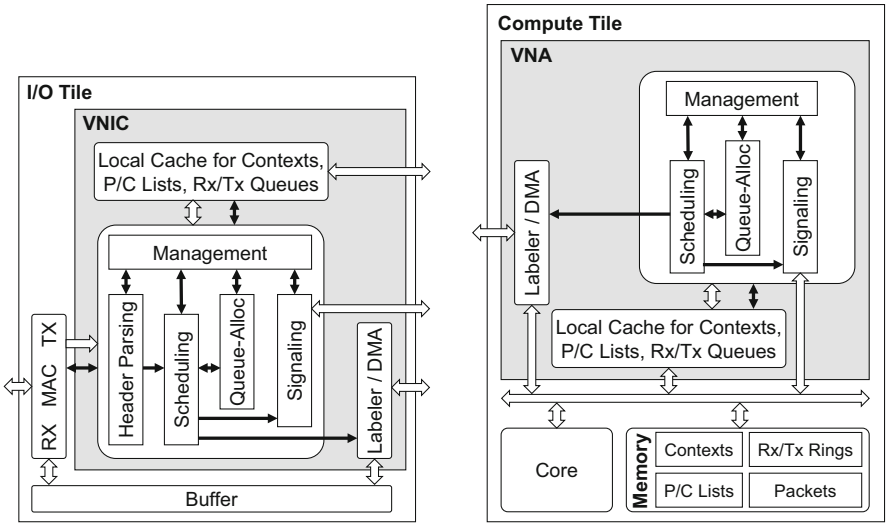


Fig. 19 VNIC and VNA architectures

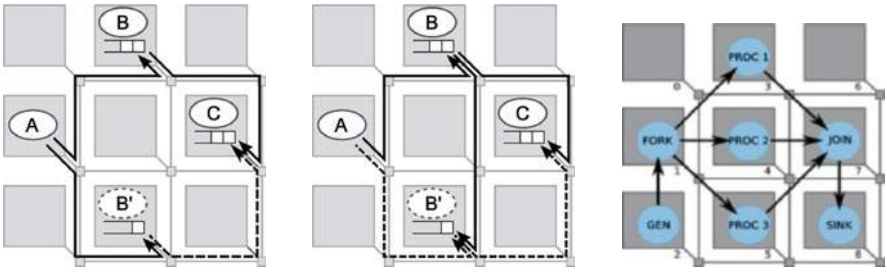
switched over to a different receiving compute tile/processor core. Depending on the migration target of T1 the incoming data will be sent to the tile executing the new T1' [8]. In Sect. 6.2 specific protocols are described to reduce downtime of tasks during migration.

To implement this helper function, both a virtualized NoC adapter (VNA) and a virtualized network interface controller (VNIC) are introduced that can be reconfigured in terms of logical communication endpoints when a task migration has to be performed [15].

VNA and VNIC target a compromise between high throughput and support for mixed-criticality application scenarios with high priority and best effort communication channels [11]. Both are based on a set of communicating finite state machines (FSMs) dedicated to specific sub-functions to cope with these requirements, as can be seen in Fig. 19. The partitioning into different FSMs enables the parallel processing of concurrent transactions in a pipelined manner.

6.2 *Advanced Communication Reconfiguration Using Protection Switching*

The common, straight-forward method for task relocation is Stop and Resume: Here, first the incoming channels of the task to be migrated are suspended, then channel state together with the task state are transferred, before the channels and task are resumed at the destination. The key disadvantage is a long downtime. Therefore, an advanced communication reconfiguration using protection switching in NoCs to



(a) (b) (c)

Fig. 20 Variants for communication migration. (**a**) Dualcast. (**b**) Forwarding. (**c**) Example migration scenario

support reliable task migration is proposed [16, 22, 23], which is inspired by protection switching mechanisms from wide area transport networks. Two alternatives to migrate communication relations of a relocated task are *dualcast* and *forwarding* as shown in Fig. 20 for a migration of task B to a different location executing B'. The procedure is to first establish an additional channel to the compute tile where the task is migrated to (location of B'). This can be either done from A being the source of the channel (*dualcast*, Fig. 20a) or from B the original location of the migrated task (*forwarding*, Fig. 20b). Then it has to be ensured that the buffers at the source and destination tiles of the migration are consistent. Finally, a seamless switch-over (task and channels) takes place from the original source to the destination. This shall avoid time-costly buffer copy and channel suspend/resume operations with a focus on low-latency and reliable adaptions in the communication layer.

The different variants have been evaluated for an example migration scenario as depicted in Fig. 20c: In a processing chain consisting of 7 tasks in total, the FORK task, which receives data from a generator task and sends data to three parallel processing tasks, is migrated to tile number 0. Figure 21 shows the latencies of the depicted execution chain during the migration, which starts at $2.5 \cdot 10^6$ cycles assuming FORK is stateless. The results have been measured using an RTL implementation of the MPSoC [16]. In Fig. 21a the situation is captured for a pure software-based implementation of the migration, whereas Fig. 21b shows the situation when all functions related to handling the migration are offloaded from the processor core. In this case task execution is not inhibited by any migration overhead, which corresponds to the situation when the VNA performs the associated functionality in hardware.

As can be seen from Fig. 21b, offloading migration protocols helps to reduce application processing latency significantly for all three variants. The dualcast and forwarding variants enable a nearly unnoticeable migration of the tasks. However, the investigations in [16] show that when migrating tasks with state, the handling of the task migration itself becomes the dominant factor in the migration delays and outweighs the benefits of the advanced switching techniques.

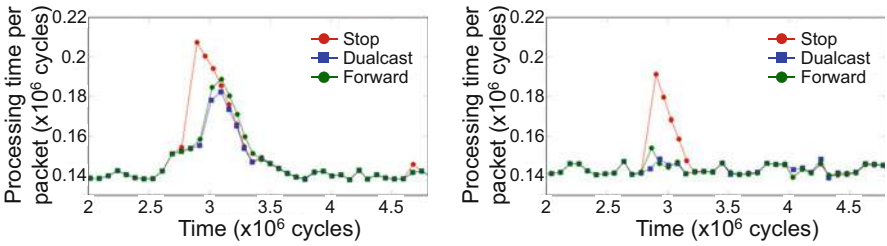


Fig. 21 Results for task migration scenarios (from [16]). (**a**) Relocation without offload. (**b**) Relocation with offload

6.3 Adaptive Modular Redundancy (AMR)

Adaptive modular redundancy (AMR) enables the dynamic establishment of a redundancy mechanism (dual or triple modular redundancy, DMR/TMR) at run-time for tasks that have a degree of criticality that may vary over time or if the operating conditions of the platform have deteriorated so that the probability of errors is too high. DMR will be used if re-execution is affordable, otherwise TMR can be applied, e.g., in case realtime requirements could not be met. AMR functionality builds upon the aforementioned services of the NoC. To establish DMR the dualcast mechanism is used and the newly established task acts as replica instead taking over the processing as in the case of migration. (For TMR two replica are established and triple-cast is applied.) Based on the running task replica, the standard mechanisms for error checking/correction and task re-execution if required are applied.

The decision to execute one or two additional replica of tasks is possibly taken as a consequence of an already impaired system reliability. On the one hand this helps to make these tasks more safe. On the other hand it increases system workload and the associated thermal load, which in turn may further aggravate the dependability issues. Therefore, this measure should be accompanied with an appropriate reliability-aware task mapping including a graceful degradation for low-critical tasks like investigated in [1]. There, the applied scheme is the following: After one of the cores exceeds a first temperature threshold T_1 a graceful degradation phase is entered. This means that tasks of high criticality are preferably assigned to cores in an exclusive manner and low-critical tasks are migrated to a "graceful degradation region" of the system. Thus, potential errors occurring in this region would involve low-critical tasks only. In a next step, if peak temperature is higher than a second threshold T_2, low-critical tasks are removed also from the graceful degradation region (NCT ejection) and are only resumed if the thermal profile allows for it.

In [1] a simulation-based investigation of this approach has been done using the Sniper simulator, McPAT and Hotspot for a 16-core Intel Xeon X5550 running SPLASH-2 and PARSEC benchmarks. Tasks have been either classified as uncritical

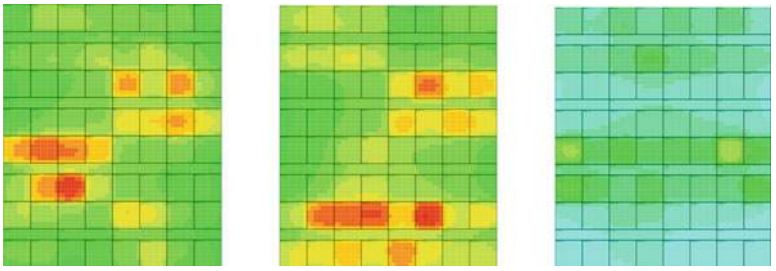


Fig. 22 Thermal profile during different phases (from [1]). The maximum system temperatures are 363 K, 378 K, and 349 K, respectively. (**a**) Initial scheduling. (**b**) Graceful degradation. (**c**) NCT ejection

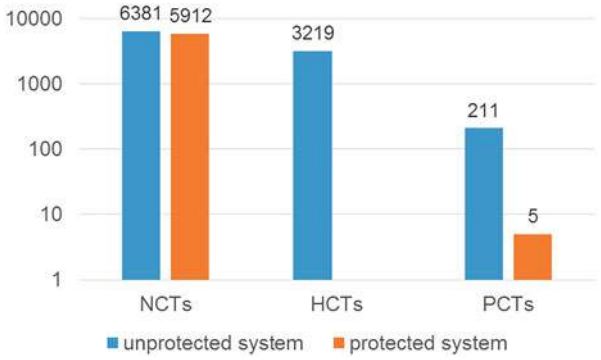


Fig. 23 Number of propagated errors per task criticality (from [1])

(NCT) or high-critical (HCT) with permanently redundant execution. As a third class, potentially critical tasks (PCT) are considered. Such tasks are dynamically replicated if the temperature of the cores they run on exceeds T_1. In the experiment, financial analysis and computer vision applications from the benchmark sets are treated as high-critical tasks (HCT). The FFT kernel as used in a wide range of applications with different criticality levels is assumed to be PCT.

In a first experiment the thermal profile has been evaluated for normal operation and the two escalating phases. As can be seen from Fig. 22 the initial thermal hotspots are relaxed at the expense of new ones in the graceful degradation region. In turn, when moving to the NCT ejection phase the chips significantly cool down.

In a further investigation, 10,000 bit-flips have been injected randomly into cache memories independent of the criticality of the tasks running on the cores. This has been done both for a system using the mechanisms described above and as a reference for a fully unprotected system.

Figure 23 shows the resulting number of propagated errors for the different task categories. In the protected system, all errors injected into cores running HCTs are

corrected, as expected. For PCTs only those errors manifest themselves in a failure that were injected when the task was not protected due to a too low temperature of the processor core. In general, not all injected errors actually lead to a failure due to masking effects in the architecture or the application memory access pattern. This can be seen for the unprotected system where the overall sum of manifested failures is less than the number of injected errors.

7 Cross-Layer

From Physics to System Level (Fig. 24) In our work, we start from the physics, where degradation effects like aging and temperature do occur. Then we analyze and investigate how these degradations alter the key transistor parameters such as threshold voltage (V_{th}), carrier mobility (μ), sub-threshold slope (SS), and drain current (I_D). Then, we study how such drift in the electrical characteristics of the transistor impacts the resilience of circuits to errors. In practice, the susceptibility to noise effects as well as to timing violations increases. Finally, we develop models for error probability that describe the ultimate impact of these degradations at the system level.

Interaction between the System Level and the Lower Abstraction Levels (Fig. 24) Running workloads at the system level induce different stress patterns for transistors and, more importantly, generate different heat over time. Temperature is one of the key stimuli when it comes to reliability degradations. Increase in temperature accelerates the underlying aging mechanisms in transistors as well as it increases the susceptibility of circuits to noise and timing violations. Such an increase in the susceptibility manifests itself as failures at the system level due to timing violations and data corruption. Therefore, different running workloads result in different probabilities of error that can be later observed at the system level.

Key Role of Management Layer The developed probability of error models helps the management layer to make proper decision. The management layer migrates the running tasks/workload from a core that starts to have a relatively higher probability of error to another "less-aged" core. Also the management layer switches this core from a high-performance mode (where high voltage and high frequency are selected leading to higher core temperatures) to a low-power mode (where low voltage and low frequency are selected leading to lower core temperatures) when it is observed that a core started to have an increase in the probability of error above an acceptable level.

Scenarios of Cross-Layer Management and existing Interdependencies In the following we demonstrate some examples of existing interdependencies between the management layer and the lower abstraction layers.

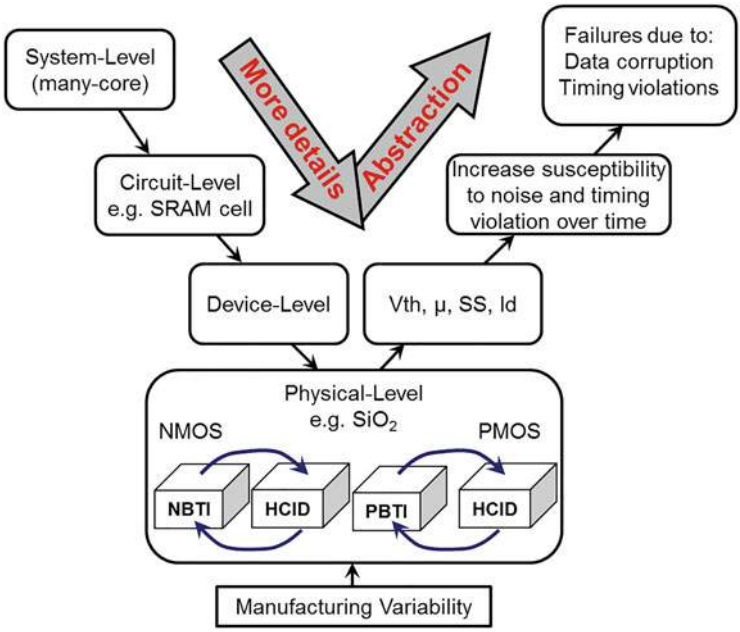


Fig. 24 Overview of how our techniques span various abstraction layers

Scenario-1: Physical and System Layer The temperature of a core increases and therefore the error probability starts to gradually rise. If a given threshold is exceeded and the core has performance margins, the first management decision would be to decrease voltage and frequency and thus limit power dissipation and in consequence counteract the temperature increase of the core.

Scenario-2: Physical, Architecture, and System Layer If there is no headroom on the core, the system management layer can now decide to migrate tasks away from that core, especially if they have high reliability requirements. Targets for migration would especially be colder, less-aged cores with a low probability of errors. With such task migrations, temperature within a system should be balanced, i.e., relieved cores can cool down, while target cores would get warmer. Further, on cores that can cool down again some of the deleterious effects start to heal, leading to a reduction in the probability of errors. In general, by continuously balancing load and as a result also temperature among cores the management layer will take care that error probabilities of cores become similar thus avoiding the situation that one core fails earlier than others. During task migrations the described support functions in the communication infrastructure (circuit layer) can be applied.

Scenario-3: Physical, Architecture, and System Layer If there is no possibility to move critical tasks to a cold core with low error probability, the management layer can employ adaptive modular redundancy (AMR) and replicate such tasks. This allows to counter the more critical operating conditions and increase reliability

by either error detection and task re-execution or by directly correcting errors when otherwise realtime requirements would not be met. However, in these cases the replica tasks will increase the overall workload of the system and thus also contribute thermal stress. In this case, dropping tasks of low criticality is a measure on system level to counter this effect.

In general, the described scenarios always form control loops starting on physical level covering temperature sensors and estimates of error probabilities and aging. They go either up to the circuit level or to the architecture/system level, where countermeasures have to be taken to prevent the system from operating under unreliable working conditions. Therefore, the mechanisms on the different abstraction levels as shown in the previous sections interact with each other and can be composed to enhance reliability in a cross-layer manner.

Further use cases tackling probabilistic fault and error modeling as well as space- and time-dependent error abstraction across different levels of the hardware/software stack of embedded systems IC components are also subject of the chapter "RAP (Resilience Articulation Point) Model."

8 Conclusion

Reliability modeling and optimization is one of the key challenges in advanced technology. With technology scaling, the susceptibility of transistors to various kinds of degradation effects induced by aging increases. As a matter of fact, temperature is the main stimulus behind aging and therefore controlling and mitigating aging can be done through a proper thermal management. Additionally, temperature itself has also a direct impact on the reliability of any circuit manifesting itself as an increase in the probability of error. In order to sustain reliability, the system level must become aware of the degradation effects occurring at the physical level and how they then propagate to higher abstraction levels all the way up to the system level. Our cross-layer approach provides the system level with accurate estimations of the probability of errors, which allows the management layer to make proper decisions to optimize the reliability. We demonstrated the existing interdependencies between the system level and lower abstraction levels and the necessity of taking them into account via cross-layer thermal management techniques.

References

1. Alouani, I., Wild, T., Herkersdorf, A., Niar, S.: Adaptive reliability for fault tolerant multicore systems. In: 2017 Euromicro Conference on Digital System Design (DSD), pp. 538–542 (2017). https://doi.org/10.1109/DSD.2017.78
2. Amrouch, H., Henkel, J.: Lucid infrared thermography of thermally-constrained processors. In: 2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), pp. 347–352. IEEE, Piscataway (2015)

3. Amrouch, H., van Santen, V.M., Ebi, T., Wenzel, V., Henkel, J.: Towards interdependencies of aging mechanisms. In: Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, pp. 478–485. IEEE, Piscataway (2014)
4. Amrouch, H., Martin-Martinez, J., van Santen, V.M., Moras, M., Rodriguez, R., Nafria, M., Henkel, J.: Connecting the physical and application level towards grasping aging effects. In: 2015 IEEE International Reliability Physics Symposium, p 3D-1. IEEE, Piscataway (2015)
5. Amrouch, H., Khaleghi, B., Gerstlauer, A., Henkel, J.: Towards aging-induced approximations. In: 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE, Piscataway (2017)
6. Amrouch, H., van Santen, V.M., Henkel, J.: Interdependencies of degradation effects and their impact on computing. IEEE Des. Test **34**(3), 59–67 (2017)
7. Boroujerdian, B., Amrouch, H., Henkel, J., Gerstlauer, A.: Trading off temperature guardbands via adaptive approximations. In: 2018 IEEE 36th International Conference on Computer Design (ICCD), pp. 202–209. IEEE, Piscataway (2018)
8. Ebi, T., Rauchfuss, H., Herkersdorf, A., Henkel, J.: Agent-based thermal management using real-time i/o communication relocation for 3d many-cores. In: Ayala, J.L., García-Cámara, B., Prieto, M., Ruggiero, M., Sicard, G. (eds.) Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation, pp. 112–121. Springer, Berlin (2011)
9. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., et al.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. circuits Syst. **32**(1), 8–23 (2012)
10. Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., et al.: Design and architectures for dependable embedded systems. In: Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 69–78. ACM, New York (2011)
11. Herkersdorf, A., Michel, H.U., Rauchfuss, H., Wild, T.: Multicore enablement for automotive cyber physical systems. Inf. Technol. **54**, 280–287 (2012). https://doi.org/10.1524/itit.2012.0690
12. Hussam, A., Jörg, H.: Evaluating and mitigating degradation effects in multimedia circuits. In: Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia '17), pp. 61–67. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3139315.3143527
13. Khdr, H., Ebi, T., Shafique, M., Amrouch, H.: mDTM: multi-objective dynamic thermal management for on-chip systems. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6. IEEE, Piscataway (2014)
14. Prakash, A., Amrouch, H., Shafique, M., Mitra, T., Henkel, J.: Improving mobile gaming performance through cooperative CPU-GPU thermal management. In: Proceedings of the 53rd Annual Design Automation Conference, p. 47. ACM, New York (2016)
15. Rauchfuss, H., Wild, T., Herkersdorf, A.: Enhanced reliability in tiled manycore architectures through transparent task relocation. In: ARCS 2012, pp. 1–6 (2012)
16. Rösch, S., Rauchfuss, H., Wallentowitz, S., Wild, T., Herkersdorf, A.: MPSoC application resilience by hardware-assisted communication virtualization. Microelectron. Reliab. **61**, 11–16 (2016). https://doi.org/10.1016/j.microrel.2016.02.009. http://www.sciencedirect.com/science/article/pii/S0026271416300282. SI: ICMAT 2015
17. van Santen, V.M., Amrouch, H., Parihar, N., Mahapatra, S., Henkel, J.: Aging-aware voltage scaling. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 576–581. IEEE, Piscataway (2016)
18. van Santen, V.M., Martin-Martinez, J., Amrouch, H., Nafria, M.M., Henkel, J.: Reliability in super-and near-threshold computing: a unified model of RTN, BTI, and PV. IEEE Trans. Circuits Syst. I Regul. Pap. **65**(1), 293–306 (2018)

19. van Santen, V.M., Diaz-Fortuny, J., Amrouch, H., Martin-Martinez, J., Rodriguez, R., Castro-Lopez, R., Roca, E., Fernandez, F.V., Henkel, J., Nafria, M.: Weighted time lag plot defect parameter extraction and GPU-based BTI modeling for BTI variability. In: 2018 IEEE International Reliability Physics Symposium (IRPS), pp. P-CR.6-1–P-CR.6-6 (2018). https://doi.org/10.1109/IRPS.2018.8353659
20. van Santen, V.M., Amrouch, H., Henkel, J.: Modeling and mitigating time-dependent variability from the physical level to the circuit level. IEEE Trans. Circuits Syst. I Regul. Pap. 1–14 (2019). https://doi.org/10.1109/TCSI.2019.2898006
21. van Santen, V.M., Amrouch, H., Henkel, J.: New worst-case timing for standard cells under aging effects. IEEE Trans. Device Mater. Reliab. **19**(1), 149–158 (2019). https://doi.org/10.1109/TDMR.2019.2893017
22. Wallentowitz, S., Rösch, S., Wild, T., Herkersdorf, A., Wenzel, V., Henkel, J.: Dependable task and communication migration in tiled manycore system-on-chip. In: Proceedings of the 2014 Forum on Specification and Design Languages (FDL), vol. 978-2-9530504-9-3, pp. 1–8 (2014). https://doi.org/10.1109/FDL.2014.7119361
23. Wallentowitz, S., Tempelmeier, M., Wild, T., Herkersdorf, A.: Network-on-chip protection switching techniques for dependable task migration on an open source MPSoC platform.

Lightweight Software-Defined Error Correction for Memories

Irina Alam, Lara Dolecek, and Puneet Gupta

The key observation behind the techniques described in this chapter is that most if not all error correction techniques and codes assume that all words stored in the memory are equally likely and important. This obviously is not true due to architectural or application context. This chapter devises new coding and correction mechanisms which leverage software or architecture "side information" to dramatically reduce the cost of error correction (Fig. 1). The methodology proposed in Sect. 1 is for recovering from detected-but-uncorrectable (DUE) errors in main memories while Sects. 2 and 3 focus on lightweight correction in on-chip caches or embedded memories.

1 Software-Defined Error Correcting Codes (SDECC)

This section focuses on the concept of Software-Defined Error Correcting Codes (SDECC), a general class of techniques spanning hardware, software, and coding theory that improves the overall resilience of systems by enabling heuristic best-effort recovery from detected-but-uncorrectable errors (DUE). The key idea is to add software support to the hardware error correcting code (ECC) so that most memory DUEs can be heuristically recovered based on available *side information* (SI) from the corresponding un-corrupted cache line contents. SDECC does not degrade memory performance or energy in the common cases when either no errors or purely hardware-correctable errors occur. Yet it can significantly improve resilience in the critical case when DUEs actually do occur.

I. Alam · L. Dolecek · P. Gupta (✉)
ECE Department, UCLA, Los Angeles, CA, USA
e-mail: irina1@ucla.edu; dolecek@ee.ucla.edu; puneetg@ucla.edu

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_9

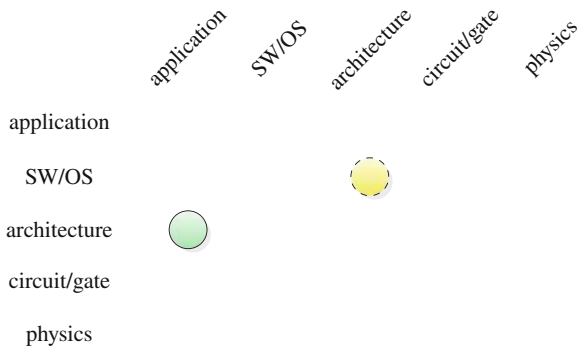


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

Details of the concepts discussed in this section can be found in the works by Gottscho et al. [11, 12].

1.1 SDECC Theory

Important terms and notation introduced here are summarized in Table 1.

A $(t)SC(t+1)SD$ *code corrects up to t symbol errors and/or detects up to (t+1) symbol errors. SDECC is based on the fundamental observation that when a* $(t+1)$*-symbol DUE occurs in a* $(t)SC(t+1)SD$ *code, there remains significant information in the received string* $\mathbf{x}$. This information can be used to recover the original message $\mathbf{m}$ with reasonable certainty.

It is not the case that the original message was completely lost, i.e., one need not naïvely choose from all q^k possible messages. If there is a $(t+1)$ DUE, there are exactly

$$N = \binom{n}{t+1}(q-1)^{(t+1)} \tag{1}$$

ways that the $(t+1)$ DUE could have corrupted the original codeword, which is less than q^k. Though a (t)SC$(t+1)$SD code can often detect more than $(t+1)$ errors, a $(t+1)$error is usually much more likely than higher bit errors. But guessing correctly out of N possibilities is still difficult. *In practice, there are just a handful of possibilities: they are referred to as* $(t+1)$DUE corrupted candidate codewords *(or candidate messages).*

Consider Fig. 2, which depicts the relationships between codewords, correctable errors (CEs), DUEs, and candidate codewords for individual DUEs for a Single-bit Error Correcting, Double-bit Error Detecting (SECDED) code. If the hardware ECC decoder registers a DUE, there can be several equidistant candidate codewords at the

Table 1 Important SDECC-specific notation

Term	Description
n	Codeword length in symbols
k	Message length in symbols
r	Parity length in symbols
b	Bits per symbol
q	Symbol alphabet size
t	Max. guaranteed correctable symbols in codeword
(t)SC$(t+1)$SD	(t)-symbol-correcting, $(t+1)$-symbol-detecting
N	Number of ways to have a DUE
μ	Mean no. of candidate codewords ∀ possible DUEs
P_G	Prob. of choosing correct codeword for a given DUE
$\overline{P_G}$	Avg. prob. of choosing correct codeword ∀ possible DUEs
d_{min}	Minimum symbol distance of code
`linesz`	Total cache line size in symbols (message content)
symbol	Logical group of bits
SECDED	Single-bit-error-correcting, double-bit-error-detecting
DECTED	Double-bit-error-correcting, triple-bit-error-detecting
SSCDSD	Single-symbol-error-correcting, double-symbol-error-detecting
ChipKill-correct	ECC construction and mem. organization that either corrects up to 1 DRAM chip failure or detects 2 chip failures

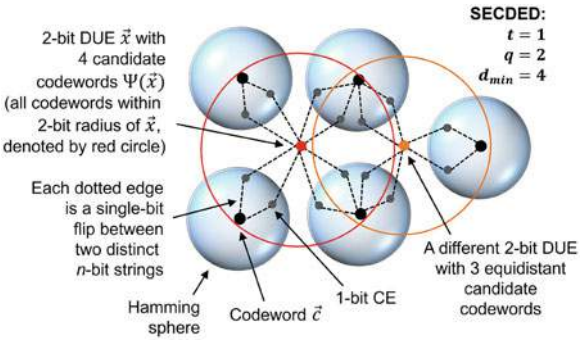


Fig. 2 Illustration of candidate codewords for 2-bit DUEs in the imaginary 2D-represented Hamming space of a binary SECDED code ($t = 1, q = 2, d_{min} = 4$). The actual Hamming space has n dimensions

q-ary Hamming distance of exactly $(t + 1)$ from the received string $\mathbf{x}$. Without any *side information* (SI) about message probabilities, under conventional principles, each candidate codeword is assumed to be equally likely. However, in the specific case of DUEs, not all messages are equally likely to occur: this allows to leverage SI about memory contents to help choose the right candidate codeword in the event of a given DUE.

1.1.1 Computing the List of Candidates

The number of candidate codewords for any given $(t+1)$ DUE $\mathbf{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice [12]. The candidate codewords for any $(t+1)$-symbol DUE received string $\mathbf{x}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\mathbf{x}$. This list depends on the error $\mathbf{e}$ and original codeword $\mathbf{c}$, but only the received string $\mathbf{x}$ is known. Fortunately, there is a simple and intuitive algorithm to find the list of candidate codewords with runtime complexity $O(nq/t)$. The detailed algorithm can be found in [12]. The essential idea is to try every possible single symbol *perturbation* $\mathbf{p}$ on the received string. Each *perturbed string* $\mathbf{y} = \mathbf{x} + \mathbf{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix $\mathbf{H}$ ($O(rn\log q)$ bits of storage). Any $\mathbf{y}$ characterized as a CE produces a candidate codeword from the decoder output and added to the list (if not already present in the list).

1.1.2 SDECC Analysis of Existing ECCs

Code constructions exhibit structural properties that affect the number of candidate codewords. In fact, distinct code constructions with the same $[n, k, d_{min}]_q$ parameters can have different values of μ and distributions of the number of candidate codewords. μ depends on the total number of minimum weight non-$\overrightarrow{0}$ codewords [12].

The SDECC theory is applied to seven code constructions of interest: SECDED, DECTED, and SSCDSD (ChipKill-Correct) constructions with typical message lengths of 64, and 128 bits. Table 2 lists properties that have been derived for each of them. Most importantly, the final column lists $\overline{P_G}$—the average (*random* baseline) probability of choosing correct codeword without SI for all possible DUEs. These probabilities are far higher than the naïve approaches of guessing randomly from q^k possible messages or from the N possible ways to have a DUE. Thus, SDECC can handle DUEs in a more optimistic way than conventional ECC approaches.

Table 2 Summary of code properties—$\overline{P_G}$ is most important for SDECC

Class of code	Code params. $[n, k, d_{min}]_q$	Type of code	Class of DUE $(t+1)$	Avg. # Cand. μ	Prob. Rcov. $\overline{P_G}$
32-bit SECDED	$[39, 32, 4]_2$	Hsiao [16]	2-bit	12.04	8.50%
32-bit SECDED	$[39, 32, 4]_2$	Davydov [7]	2-bit	9.67	11.70%
64-bit SECDED	$[72, 64, 4]_2$	Hsiao [16]	2-bit	20.73	4.97%
64-bit SECDED	$[72, 64, 4]_2$	Davydov [7]	2-bit	16.62	6.85%
32-bit DECTED	$[45, 32, 6]_2$	–	3-bit	4.12	28.20%
64-bit DECTED	$[79, 64, 6]_2$	–	3-bit	5.40	20.53%
128-bit SSCDSD	$[36, 32, 4]_{16}$	Kaneda [17]	2-sym.	3.38	39.88%

1.2 SDECC Architecture

SDECC consists of both hardware and software components to enable recovery from DUEs in main memory DRAM. A simple hardware/software architecture whose block diagram is depicted in Fig. 3 can be used. Although the software flow includes an instruction recovery policy, it is not presented in this chapter because DUEs on instruction fetches are likely to affect clean pages that can be remedied using a page fault (as shown in the figure).

The key addition to hardware is the *Penalty Box*: a small buffer in the memory controller that can store each codeword from a cache line (shown on the left-hand side of Fig. 3). When a memory DUE occurs, hardware stores information about the error in the Penalty Box and raises an error-reporting interrupt to system software. System software then reads the Penalty Box, derives additional context about the error—and using basic coding theory and knowledge of the ECC implementation—quickly computes a list of all possible *candidate messages*, one of which is guaranteed to match the original information that was corrupted by the DUE. A software-defined data recovery policy heuristically recovers the DUE in a best-effort manner by choosing the most likely remaining candidate based on available *side information* (SI) from the corresponding un-corrupted cache line contents; if confidence is low, the policy instead forces a panic to minimize the risk of accidentally induced mis-corrected errors (MCEs) that result in intolerable non-silent data corruption (NSDC). Finally, system software writes back the *recovery target* message to the Penalty Box, which allows hardware to complete the afflicted memory read operation.

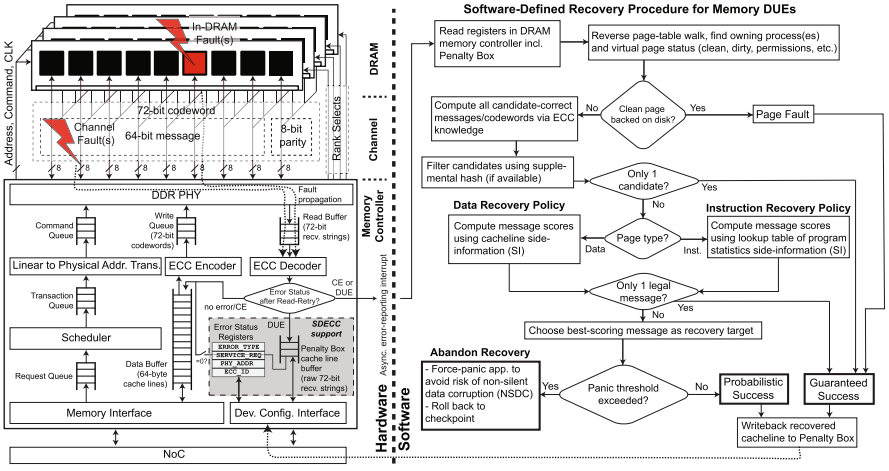


Fig. 3 Block diagram of a general hardware and software implementation of SDECC. The figure depicts a typical DDRx-based main memory subsystem with 64-byte cache lines, x8 DRAM chips, and a $[72, 64, 4]_2$ SECDED ECC code. Hardware support necessary to enable SDECC is shaded in gray. The instruction recovery policy is outside the scope of this work [12]

Overheads The area and power overhead of the essential SDECC hardware support is negligible. The area required per Penalty Box is approximately 736 μm^2 when synthesized with 15 nm Nangate technology—this is approximately one millionth of the total die area for a 14 nm Intel Broadwell-EP server processor [9]. The SDECC design incurs no latency or bandwidth overheads for the vast majority of memory accesses where no DUEs occur. This is because the Penalty Box and error-reporting interrupt are not on the critical path of memory accesses. When a DUE occurs, the latency of the handler and recovery policy is negligible compared to the expected mean time between DUEs or typical checkpoint interval of several hours.

1.3 Data Recovery Policy

In this section, recovery of DUEs in data (i.e., memory reads due to processor loads) is discussed because they are more vulnerable than DUEs in instructions as mentioned before. Possible recovery policies for instruction memory have been discussed in [11]. There are potentially many sources of SI for recovering DUEs in data. Based on the notion of *data similarity*, a simple but effective data recovery policy called *Entropy-Z* is discussed here that chooses the candidate that minimizes overall cache line Shannon entropy.

1.3.1 Observations on Data Similarity

Entropy is one of the most powerful metrics to measure data similarity. Two general observations can be made about the prevalence of low data entropy in memory.

- **Observation 1.** There are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte, halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs).
- **Observation 2.** In addition to spatial and temporal locality in their memory access patterns, applications have inherent *value locality* in their data, regardless of their hardware representation. For example, an image-processing program is likely to work on regions of pixels that exhibit similar color and brightness, while a natural language processing application will see certain characters and words more often than others.

Similar observations have been made to compress memory [2, 18, 24, 26, 28, 35] and to predict [20] or approximate processor load values [22, 23, 36]. Low byte-granularity intra-cache line entropy is observed throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite. Let $P(X)$ be the normalized relative frequency distribution of a `linesz`$\times b$-bit cache line that has been carved

into equal-sized Z-bit symbols, where each symbol χ_i can take 2^Z possible values.[1] Then the Z-bit-granularity `entropy` is computed as follows:

$$\texttt{entropy} = -\sum_{i=1}^{\texttt{linesz}\times b/Z} P(\chi_i) log_2 P(\chi_i). \tag{2}$$

The average intra-cacheline byte-level entropy of the SPEC CPU2006 suite was found to be 2.98 bits (roughly half of maximum).

These observations can be leveraged using the data recovery policy *Entropy-Z Policy*. With this policy, SDECC first computes the list of candidate messages using the algorithm described in Sect. 1.1.1 and extracts the cache line side information. Each candidate message is then inserted into appropriate position in the affected cache line and the entropy is computed using Eq. 2. The policy then chooses the candidate message that minimizes overall cache line entropy. The chance that the policy chooses the wrong candidate message is significantly reduced by deliberately forcing a *panic* whenever there is a tie for minimum entropy or if the mean cache line entropy is above a specified threshold `PanicThreshold`. The downside to this approach is that some forced panics will be false positives, i.e., they would have otherwise recovered correctly.

In the rest of the chapter, unless otherwise specified, $Z = 8$ bits, $\texttt{linesz}\times b = 512$ bits and $\texttt{PanicThreshold} = 4.5$ bits (75% of maximum entropy) are used, which were determine to work well across a range of applications. Additionally, the *Entropy-8* policy performs very well compared to several alternatives.

1.4 Reliability Evaluation

The impact of SDECC is evaluated on system-level reliability through a comprehensive error injection study on memory access traces. The objective is to estimate the fraction of DUEs in memory that can be recovered correctly using the SDECC architecture and policies while ensuring a minimal risk of MCEs.

1.4.1 Methodology

The SPEC CPU2006 benchmarks are compiled against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [34] using the official tools [25]. Each benchmark is executed on top of the RISC-V proxy kernel [32] using the Spike simulator [33] that was modified to produce representative memory access

[1] Entropy symbols are not to be confused with the codeword symbols, which can also be a different size.

traces. Each trace consists of randomly sampled 64-byte demand read cache lines, with an average interval between samples of one million accesses.

Each trace is analyzed offline using a MATLAB model of SDECC. For each benchmark and ECC code, 1000 q-ary messages from the trace were chosen randomly and encoded, and were injected with $min(1000, N)$ randomly sampled $(t + 1)$-symbol DUEs. N here is the number of ways to have a DUE. For each codeword/error pattern combination, the list of candidate codewords was computed and the data recovery policy was applied. A *successful recovery* occurs when the policy selects a candidate message that matches the original; otherwise, the policy either causes a *forced panic* or recovery fails by accidentally inducing an MCE. Variability in the reported results is negligible over many millions of individual experiments.

Note that the *absolute* error magnitudes for DUEs and SDECC's impact on *overall* reliability should not be compared directly between codes with distinct $[n, k, d_{min}]_q$ (e.g., a double-bit error for SECDED is very different from a double-chip DUE for ChipKill). Rather, what matters most is the *relative* fraction of DUEs that can be saved using SDECC for a given ECC code.

Entropy-8 is exclusively used as the data recovery policy in all the evaluations. This is because when the raw successful recovery rates of six different policies for three ECCs without including any forced panics were compared, *Entropy-8* performed the best [12]. Few examples of alternate policies include *Entropy-Z* policy variants with $Z = 4$ and $Z = 16$ and *Hamming* which chooses the candidate that minimizes the average binary Hamming distance to the neighboring words in the cacheline. The 8-bit entropy symbol size performs best because its alphabet size ($2^8 = 256$ values) matches well with the number of entropy symbols per cacheline (64) and with the byte-addressable memory organization. For instance, both *Entropy-4* and *Entropy-16* do worse than *Entropy-8* because the entropy symbol size results in too many aliases at the cacheline level and because the larger symbol size is less efficient, respectively.

1.4.2 Recovery Breakdown

SDECC is evaluated next for each ECC using its conventional form, to understand the impact of the recovery policy's *(Entropy-8)* forced panics on the successful recovery rate and the MCE rate. The overall results with forced panics *taken* (main results, gray cell shading) and *not taken* are shown in Table 3.

There are two baseline DUE recovery policies: *conventional* (always panic for every DUE) and *random* (choose a candidate randomly, i.e., $\overline{P_G}$). It is observed that when panics are taken the MCE rate drops significantly by a factor of up to 7.3× without significantly reducing the success rate. This indicates that the `PanicThreshold` mechanism appropriately judges when SDECC is unlikely to correctly recover the original information.

These results also show the impact of code construction on successes, panics, and MCEs. When there are fewer average candidates μ then the chances of successfully

Table 3 Percent Breakdown of SDECC *Entropy-8* Policy (M = MCE, P = forced panic, S = success) [12]

	Panics taken			Panics not taken			*Random* baseline		
	M	P	S	M	P	S	M	P	S
Conv. baseline	–	100	–						
$[39, 32, 4]_2$ Hsiao	5.3	25.6	69.1	27.3	–	72.7	91.5	–	8.5
$[39, 32, 4]_2$ Davydov	4.5	25.2	70.3	24.0	–	76.0	88.3	–	11.7
$[72, 64, 4]_2$ Hsiao	4.7	23.7	71.6	24.7	–	75.3	95.0	–	5.0
$[72, 64, 4]_2$ Davydov	4.1	21.9	74.0	22.3	–	77.7	93.2	–	6.9
$[45, 32, 6]_2$ DECTED	2.2	20.3	77.5	14.5	–	85.5	71.8	–	28.2
$[79, 64, 6]_2$ DECTED	1.5	14.5	84.0	11.0	–	89.0	79.5	–	20.5
$[36, 32, 4]_{16}$ SSCDSD	1.5	12.8	85.7	8.5	–	91.5	60.1	–	39.9

recovering are much higher than that of inducing MCEs. The $[72, 64, 4]_2$ SECDED constructions perform similarly to their $[39, 32, 4]_2$ variants even though the former have lower baseline $\overline{P_G}$. This is a consequence of the *Entropy-8* policy: larger n combined with lower μ provides the greatest opportunity to differentiate candidates with respect to overall intra-cacheline entropy. For the same n, however, the effect of SECDED construction is more apparent. The Davydov codes recover about 3–4% more frequently than their Hsiao counterparts when panics are not taken (similar to the baseline improvement in $\overline{P_G}$). When panics are taken, however, the differences in construction are less apparent because the policy `PanicThreshold` does not take into account Davydov's typically lower number of candidates.

The breakdown between successes, panics, and MCEs is examined in more detail. Figure 4 depicts the DUE recovery breakdowns for each ECC construction and SPEC CPU2006 benchmark when forced panics are taken. Figure 4a shows the fraction of DUEs that result in success (black), panics (gray), and MCEs (white). Figure 4b further breaks down the forced panics (gray from Fig. 4a) into a fraction that are *false positive* (light purple, and would have otherwise been correct) and others that are *true positive* (dark blue, and managed to avoid an MCE). Each cluster of seven stacked bars corresponds to the seven ECC constructions.

It can be seen that much lower MCE rates are achieved than the *random* baseline yet also panic much less often than the *conventional* baseline for all benchmarks, as shown in Fig. 4a. This policy performs best on integer benchmarks due to their lower average intra-cacheline entropy. For certain floating-point benchmarks, however, there are many forced panics because they frequently have high data entropy above `PanicThreshold`. A `PanicThreshold` of 4.5 bits for these cases errs on the side of caution as indicated by the false positive panic rate, which can be up to 50%. Without more side information, for high-entropy benchmarks, it would be difficult for any alternative policy to frequently recover the original information with a low MCE rate and few false positive panics.

With almost no hardware overheads, SDECC used with SSCDSD ChipKill can recover correctly from up to 85.7% of double-chip DUEs while eliminating 87.2% of would-be panics; this could improve system availability considerably. However,

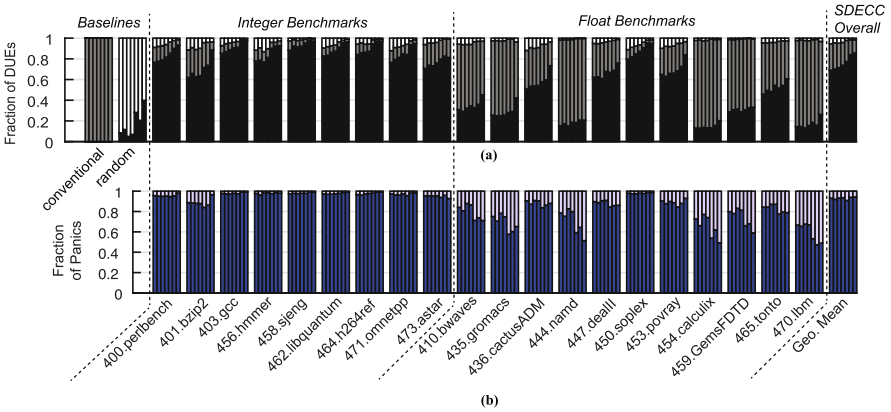


Fig. 4 Detailed breakdown of DUE recovery results when forced panics are taken. Results are shown for all seven ECC constructions, listed left to right within each cluster: $[39, 32, 4]_2$ Hsiao SECDED—$[39, 32, 4]_2$ Davydov SECDED—$[72, 64, 4]_2$ Hsiao SECDED—$[72, 64, 4]_2$ Davydov SECDED—$[45, 32, 6]_2$ DECTED—$[79, 64, 6]_2$ DECTED—$[36, 32, 4]_{16}$ SSCDSD ChipKill-Correct. (**a**) Recovery breakdown for the *Entropy-8* policy, where each DUE can result in an unsuccessful recovery causing an MCE (white), forced panic (gray), or successful recovery (black). (**b**) Breakdown of forced panics (gray bars in (**a**)). A true positive panic (dark blue) successfully mitigated a MCE, while a false positive panic (light purple) was too conservative and thwarted an otherwise-successful recovery [12]

SDECC with ChipKill introduces a 1% risk of converting a DUE to an MCE. Without further action taken to mitigate MCEs, this small risk may be unacceptable when application correctness is of paramount importance.

2 Software-Defined Error-Localizing Codes (SDELC): Lightweight Recovery from Soft Faults at Runtime

For embedded memories, it is always challenging to address reliability concerns as additional area, power, and latency overheads of reliability techniques need to be minimized as much as possible. *Software-Defined Error-Localizing Codes* (SDELC) is a hybrid hardware/software technique that deals with single-bit soft faults at runtime using novel *Ultra-Lightweight Error-Localizing Codes* (UL-ELC) with a software-defined error handler that knows about the UL-ELC construction and implements a heuristic recovery policy. UL-ELC codes are stronger than basic single-error detecting (SED) parity, yet they have lower storage overheads than a single-error-correcting (SEC) Hamming code. Like SED, UL-ELC codes can detect single-bit errors, yet they can additionally *localize* them to a *chunk* of the erroneous codeword. UL-ELC codes can be explicitly designed such that chunks align with meaningful message context, such as the fields of an encoded instruction.

SDELC then relies on *side information* (SI) about application memory contents to heuristically recover from the single-bit fault. Unlike the general-purpose Software-Defined ECC (SDECC), SDELC focuses on heuristic error recovery that is suitable for microcontroller-class IoT devices.

Details of the concepts discussed in this section can be found in the work by Gottscho et al. [13].

2.1 Ultra-Lightweight Error-Localizing Codes (UL-ELC)

In today's systems, either basic SED parity is used to detect random single-bit errors or a Hamming SEC code is used to correct them. Unfortunately, Hamming codes are expensive for small embedded memories: they require six bits of parity per memory word size of 32 bits (an 18.75% storage overhead). On the other hand, basic parity only adds one bit per word (3.125% storage overhead), but without assistance by other techniques it cannot correct any errors.

Localizing an error is more useful than simply detecting it. If the error is localized to a *chunk* of length ℓ bits, there are only ℓ *candidate codewords* for which a single-bit error could have produced the received (corrupted) codeword. A naïve way of localizing a single-bit error to a particular chunk is to use a trivial segmented parity code, i.e., assign a dedicated parity bit to each chunk. However, this method is very inefficient because to create C chunks C parity bits are needed: essentially, split up the memory words into smaller pieces.

Instead *Ultra-Lightweight* ELCs (UL-ELCs) is simple and customizable—given r redundant parity bits—it can localize any single-bit error to one of $C = 2^r - 1$ possible chunks. This is because there are $2^r - 1$ distinct non-zero columns that can be used to form the parity-check matrix $\mathbf{H}$ for the UL-ELC (for single-bit errors, the error syndrome is simply one of the columns of $\mathbf{H}$). To create a UL-ELC code, a distinct non-zero binary column vector of length r bits is assigned to each chunk. Then each column of $\mathbf{H}$ is simply filled in with the corresponding chunk vector. Note that r of the chunks will also contain the associated parity bit within the chunk itself and are called *shared chunks*, and they are precisely the chunks whose columns in $\mathbf{H}$ have a Hamming weight of 1. Since there are r shared chunks, there must be $2^r - r - 1$ *unshared chunks*, which each consist of only data bits. Shared chunks are unavoidable because the parity bits must also be protected against faults, just like the message bits.

An UL-ELC code has a minimum distance of two bits by construction to support detection and localization of single-bit errors. Thus, the set of candidate codewords must also be separated from each other by a Hamming distance of exactly two bits. (A minimum codeword distance of two bits is required for SED, while three bits are needed for SEC, etc.)

For *an example* of an UL-ELC construction, consider the following $\mathbf{H}_{\text{example}}$ parity-check matrix with nine message bits and $r = 3$ parity bits:

$$\mathbf{H}_{\text{example}} = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} S_1 & S_2 & S_3 & S_4 & S_4 & S_5 & S_6 & S_6 & S_7 & S_5 & S_6 & S_7 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array},$$

where d_i represents the ith data bit, p_j is the jth redundant parity bit, c_k is the kth parity-check equation, and S_l enumerates the distinct error-localizing chunk that a given bit belongs to. Because $r = 3$, there are $N = 7$ chunks. Bits $d_1, d_2,$ and d_3 each have the SEC property because no other bits are in their respective chunks. Bits d_4 and d_5 make up an unshared chunk S_4 because no parity bits are included in S_4. The remaining data bits belong to shared chunks because each of them also includes at least one parity bit. Notice that any data or parity bits that belong to the same chunk S_l have identical columns of $\mathbf{H}$, e.g., d_7, d_8, and p_2 all belong to S_6 and have the column [0; 1; 0].

The two key properties of UL-ELC (that do not apply to generalized ELC codes) are: (1) the length of the data message is independent of r and (2) each chunk can be an arbitrary length. The freedom to choose the length of the code and chunk sizes allows the UL-ELC design to be highly adaptable. Additionally, UL-ELC codes can offer SEC protection on up to $2^r - r - 1$ selected message bits by having the unshared chunks each correspond to a single data bit.

2.2 Recovering SEUs in Instruction Memory

This section focuses on an UL-ELC construction and recovery policy for dealing with single-bit soft faults in instruction memory. The code and policy are jointly crafted to exploit SI about the ISA itself. This SDELC implementation example targets the open-source and free 64-bit RISC-V (RV64G) ISA [34], but the approach is general and could apply to any other fixed-length or variable-length RISC or CISC ISA. Note that although RISC-V is actually a little-endian architecture, for sake of clarity big-endian is used in this example.

The UL-ELC construction for instruction memory has seven chunks that align to the finest-grain boundaries of the different fields in the RISC-V codecs. These codecs, the chunk assignments, and the complete parity-check matrix $\mathbf{H}$ are shown in Table 4. The `opcode`, `rd`, `funct3`, and `rs1` fields are the most commonly used—and potentially the most critical—among the possible instruction encodings, so each of them is assigned a dedicated chunk that is unshared with the parity bits. The fields which vary more among encodings are assigned to the remaining three shared chunks, as shown in the figure. The recovery policy can thus distinguish

Table 4 Proposed 7-Chunk UL-ELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0)

bit →	31 27	26 25	24 20	19 15	14 12	11 7	6 0	-1 -3
Type-UJ	imm[20|10:1|11|19:12]					rd	opcode	parity
Type-U	imm[31:12]					rd	opcode	parity
Type-I	imm[11:0]			rs1	funct3	rd	opcode	parity
Type-R	funct7		rs2	rs1	funct3	rd	opcode	parity
Type-S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	parity
Type-SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	parity
Type-R4	rs3	funct2	rs2	rs1	funct3	rd	opcode	parity

Chunk	C_1 (shared)	C_2 (shared)	C_3 (shared)	C_4	C_5	C_6	C_7	C_3	C_2	C_1
Parity-	11111	00	00000	11111	000	11111	1111111	1	0	0
Check	00000	11	00000	00000	111	11111	1111111	0	1	0
H	00000	00	11111	11111	111	00000	1111111	0	0	1

the impact of an error in different parts of the instruction. For example, when a fault affects shared chunk C_1, the fault is either in one of the five MSBs of the instruction, or in the last parity bit. Conversely, when a fault is localized to unshared chunk C_7 in Table 4, the UL-ELC decoder can be certain that the `opcode` field has been corrupted.

The instruction recovery policy consists of three steps.

- **Step 1.** A software-implemented instruction decoder is applied to filter out any candidate messages that are illegal instructions. Most bit patterns decode to illegal instructions in three RISC ISAs that were characterized: 92.33% for RISC-V, 72.44% for MIPS, and 66.87% for Alpha. This can be used to dramatically improve the chances of a successful SDELC recovery.
- **Step 2.** Next, the probability of each valid message is estimated using a small pre-computed lookup table that contains the relative frequency that each instruction appears. The relative frequencies of legal instructions in most applications follow power-law distribution [13]. This is used to favor more common instructions.
- **Step 3.** The instruction that is most common according to the SI lookup table is chosen. In the event of a tie, the instruction with the longest leading-pad of 0s or 1s is chosen. This is because in many instructions, the MSBs represent immediate values (as shown in Table 4). These MSBs are usually low-magnitude signed integers or they represent 0-dominant function codes.

If the SI is strong, then there is normally a higher chance of correcting the error by choosing the right candidate.

2.3 Recovering SEUs in Data Memory

In general-purpose embedded applications, data may come in many different types and structures. Because there is no single common data type and layout in memory, evenly spaced UL-ELC constructions can be used and the software trap handler can be granted additional control about how to recover from errors, similar to the general idea from SuperGlue [31].

The SDELC recovery support can be built into the embedded application as a small C library. The application can push and pop custom SDELC error handler functions onto a registration stack. The handlers are defined within the scope of a subroutine and optionally any of its callees and can define specific recovery behaviors depending on the context at the time of error. Applications can also enable and disable recovery at will.

When the application does not disable recovery nor specify a custom behavior, all data memory errors are recovered using a default error handler implemented by the library. The default handler computes the average Hamming distance to nearby data in the same 64-byte chunk of memory (similar to taking the intra-cache line distance in cache-based systems). The candidate with the minimum average

Hamming distance is selected. This policy is based on the observation that spatially local and/or temporally local data tends to also be correlated, i.e., it exhibits *value locality* [20].

The application-defined error handler can specify recovery rules for individual variables within the scope of the registered subroutine. They include globals, heap, and stack-allocated data. This is implemented by taking the runtime address of each variable requiring special handling. For instance, an application may wish critical data structures to never be recovered heuristically; for these, the application can choose to force a crash whenever a soft error impacts their memory addresses. The SDELC library support can increase system reliability, but the programmer is required to spend effort annotating source code for error recovery. This is similar to annotation-based approaches taken by others for various purposes [4, 5, 10, 21, 29, 37].

2.4 SDELC Architecture

The SDELC architecture is illustrated in Fig. 5 for a system with split on-chip instruction and data scratchpad memories (SPMs) (each with its own UL-ELC code) and a single-issue core that has an in-order pipeline.

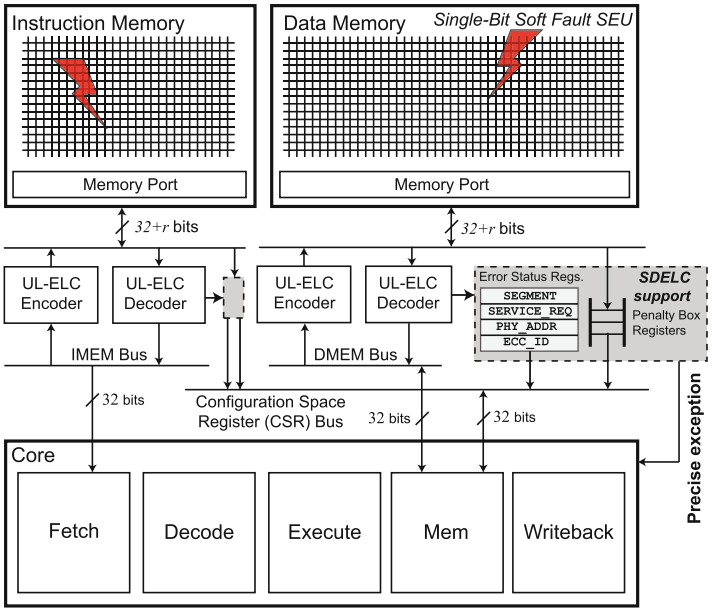


Fig. 5 Architectural support for SDELC on an microcontroller-class embedded system

When a codeword containing a single-bit soft fault is read, the UL-ELC decoder detects and localizes the error to a specific chunk of the codeword and places error information in a *Penalty Box* register (shaded in gray in the figure). A precise exception is then generated, and software traps to a handler that implements the appropriate SDELC recovery policy for instructions or data.

Once the trap handler has decided on a candidate codeword for recovery, it must correctly commit the state in the system such that it appears *as if* there was no memory control flow disruption. For instruction errors, because the error occurred during a fetch, the program counter (`pc`) has not yet advanced. To complete the trap handler, the candidate codeword is written back to instruction memory. If it is not accessible by the load/store unit, one could use hardware debug support such as JTAG. The previously trapped instruction is re-executed after returning from the trap handler, which will then cause the `pc` to advance and re-fetch the instruction that had been corrupted by the soft error. On the other hand, data errors are triggered from the memory pipeline stage by executing a load instruction. The chosen candidate codeword is written back to data memory to scrub the error, the register file is updated appropriately, and `pc` is manually advanced before returning from the trap handler.

2.5 Soft Fault Recovery Using SDELC

To evaluate SDELC, Spike was modified to produce representative memory access traces of 11 benchmarks as they run to completion. Five benchmarks are `blowfish` and `sha` from the MiBench suite [14] as well as `dhrystone`, `matmulti`, and `whetstone`. The remaining six benchmarks were added from the AxBench approximate computing C/C++ suite [37]: `blackscholes`, `fft`, `inversek2j`, `jmeint`, `jpeg`, and `sobel`. Each trace was analyzed offline using a MATLAB model of SDELC. For each workload, 1000 instruction fetches and 1000 data reads were randomly selected from the trace and exhaustively all possible single-bit faults were applied to each of them.

SDELC recovery of the random soft faults was evaluated using three different UL-ELC codes ($r = 1, 2, 3$). Recall that the $r = 1$ code is simply a single parity bit, resulting in 33 candidate codewords. (For basic parity, there are 32 message bits and one parity bit, so there are 33 ways to have had a single-bit error.) For the data memory, the UL-ELC codes were designed with the chunks being equally sized: for $r = 2$, there are either 11 or 12 candidates depending on the fault position (34 bits divided into three chunks), while for $r = 3$ there are always five candidates (35 bits divided into seven chunks). For the instruction memory, chunks are aligned to important field divisions in the RV64G ISA. Chunks for the $r = 2$ UL-ELC construction match the fields of the Type-U instruction codecs (the `opcode` being the unshared chunk). Chunks for the $r = 3$ UL-ELC code align with fields in the Type-R4 codec (as presented in Table 4). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting.

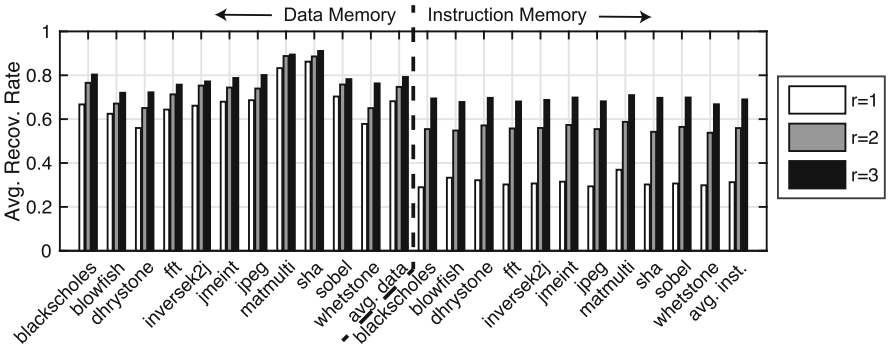


Fig. 6 Average rate of recovery using SDELC from single-bit soft faults in instruction and data memory. r is the number of parity bits in the UL-ELC construction

2.5.1 Overall Results

The overall SDELC results are presented in Fig. 6. The recovery rates are relatively consistent over each benchmark, especially for instruction memory faults, providing evidence of the general efficacy of SDELC. One important distinction between the memory types is the sensitivity to the number r of redundant parity bits per message. For the data memory, the simple $r = 1$ parity yielded surprisingly high rates of recovery using our policy (an average of 68.2%). Setting r to three parity bits increases the average recovery rate to 79.2% thanks to fewer and more localized candidates to choose from. On the other hand, for the instruction memory, the average rate of recovery increased from 31.3% with a single parity bit to 69.0% with three bits.

These results are a significant improvement over a guaranteed system crash as is traditionally done upon error detection using single-bit parity. Moreover, these results are achieved using no more than half the overhead of a Hamming SEC code, which can be a significant cost savings for small IoT devices. Based on these results, using $r = 1$ parity for data seems reasonable, while $r = 3$ UL-ELC constructions can be used to achieve 70% recovery for both memories with minimal overhead.

3 Parity++ : Lightweight Error Correction for Last Level Caches and Embedded Memories

This section focuses on another novel lightweight error correcting code—Parity++: a novel lightweight unequal message protection scheme for last level caches or embedded memories that preferentially provides stronger error protection to certain "special messages." As the name suggests, this coding scheme requires one extra bit above a simple parity Single-bit Error Detection (SED) code while providing SED

for all messages and Single-bit Error Correction (SEC) for a subset of messages. Thus, it is stronger than just basic SED parity and has much lower parity storage overhead (3.5× and 4× lower for 32-bit and 64-bit memories, respectively) than a traditional Single-bit Error Correcting, Double-bit Error Detecting (SECDED) code. Error detection circuitry often lies on the critical path and is generally more critical than error correction circuitry as error occurrences are rare even with an increasing soft error rate. This coding scheme has a much simpler error detection circuitry that incurs lower energy and latency costs than the traditional SECDED code. Thus, Parity++ is a lightweight ECC code that is ideal for large capacity last level caches or lightweight embedded memories. Parity++ is also evaluated with a memory speculation procedure [8] that can be generally applied to any ECC protected cache to hide the decoding latency while reading messages when there are no errors.

Details of the concepts discussed in this section can be found in the work by Alam et al. [1] and Schoeny et al. [30].

3.1 Application Characteristics

As mentioned in Sects. 1.3 and 2.2, data in applications is generally very structured and instructions mostly follow power-law distribution. This means most instructions in the memory would have the same opcode. Similarly, the data in the memory is usually low-magnitude signed data of a certain data type. However, these values get represented inefficiently, for e.g., 4-byte integer type used to represent values that usually need only 1-byte. Thus, in most cases, the MSBs would be a leading-pad of 0s or 1s. The approach of utilizing these characteristics in applications complements recent research on data compression in cache and main memory systems such as frequent value/pattern compression [3, 35], base-delta-immediate compression [27], and bit-plane compression [19]. However, the main goal here is to provide stronger error protection to these special messages that are chosen based on the knowledge of data patterns in context.

3.2 Parity++ Theory

Parity++ is a type of *unequal message protection* code, in that specific messages are designated a priori to have extra protection against errors as shown in Fig. 7. As in [30], there are two classes of messages, normal and special, and they are mapped to normal and special codewords, respectively. When dealing with the importance or frequency of the underlying data, it is referred to as messages; when discussing error detection/correction capabilities it is referred to as codewords.

Codewords in Parity++ have the following error protection guarantees: normal codewords have single-error detection; special codewords have single-error cor-

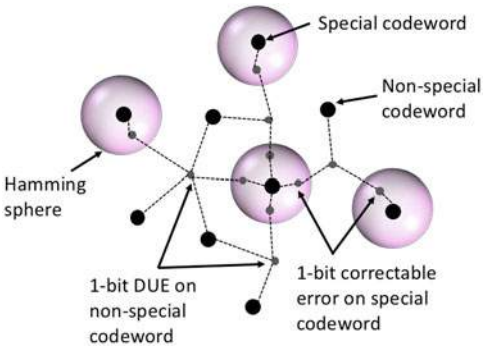


Fig. 7 Conceptual illustration of Parity++ for 1-bit error (CE = Correctable Error, DUE = Detected but Uncorrectable Error)

rection. Let us partition the codewords in the code C into two sets, $\mathcal{N}$ and $\mathcal{S}$, representing the normal and special codewords, respectively. The minimum distance properties necessary for the aforementioned error protection guarantees of Parity++ are as follows:

$$\min_{\mathbf{u},\mathbf{v}\in\mathcal{N},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 2, \tag{3}$$

$$\min_{\mathbf{u}\in\mathcal{N},\mathbf{v}\in\mathcal{S}} d_H(\mathbf{u},\mathbf{v}) \geq 3, \tag{4}$$

$$\min_{\mathbf{u},\mathbf{v}\in\mathcal{S},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 3. \tag{5}$$

A second defining characteristic of the Parity++ code is that the length of a codeword is only two bits longer than a message, i.e., $n = k + 2$. Thus, Parity++ requires only two bits of redundancy.

For the context of this work, let us assume that Parity++ always has message length k as a power of 2. The overall approach to constructing the code is to create a Hamming subcode of a SED code [15]; when an error is detected, it is decoded to the neighboring special codeword. The overall code has $d_{min} = 2$, but a block in $\mathbf{G}$, corresponding to the special messages, has $d_{min} \geq 3$. For the sake of notational convenience, let us go through the steps of constructing the (34, 32) Parity++ code (as opposed to the generic $(k + 2, k)$ Parity++ code).

The first step is to create the generating matrix for the Hamming code whose message length is at least as large as the message length in the desired Parity++ code; in this case, the (63, 57) Hamming code is used. Let α be a primitive element of GF(2^6) such that $1+x+x^6 = 0$, then the generator polynomial is simply $g_S(x) = 1 + x + x^6$ (and the generator matrix is constructed using the usual polynomial coding methods). The next step is to shorten this code to (32, 26) by expurgating and puncturing (i.e., deleting) the right and bottom 31 columns and rows. Then add a column of 1s to the end, resulting in a generator matrix, which is denoted as $\mathbf{G}_S$, for a (33, 26) code with $d_{min} = 4$.

For the next step in the construction of the generating matrix of the (34, 32) Parity++ code, $\mathbf{G_N}$ is added on top of $\mathbf{G}_S$, where $\mathbf{G_N}$ is the first 6 rows of the generator matrix using the generator polynomial $g_N(x) = 1 + x$, with an appended row of 0s at the end. Note that $\mathbf{G_N}$ is the generator polynomial of a simple parity-check code. By using this polynomial subcode construction, a generator matrix is built with overall $d_{min} = 2$, with the submatrix $\mathbf{G}_S$ having $d_{min} = 4$. At this point, notice that messages that begin with 6 0s only interact with $\mathbf{G}_S$; these messages will be the special messages. Note that Conditions 3 and 5 are satisfied; however, Condition 4 is not satisfied. To meet the requirement, a single non-linear parity bit is added that is a NOR of the bits corresponding to $\mathbf{G_N}$, in this case, the first 6 bits.

The final step is to convert $\mathbf{G}_S$ to systematic form via elementary row operations. Note that these row operations preserve all 3 of the required minimum distance properties of Parity++. As a result, the special codewords (with the exception of the known prefix) are in systematic form. For example, in the (34, 32) Parity++ code, the first 26 bits of a special codeword are simply the 26 bits in the message (not including the leading run of 6 0s).

At the encoding stage of the process, when the message is multiplied by $\mathbf{G}$, the messages denoted as special must begin with a leading run of $\log_2(k) + 1$ 0's. However, the original messages that are deemed to be special do not have to follow this pattern as one can simply apply a pre-mapping before the encoding step, and a post-mapping after the decoding step.

In the $(34, 32)$ Parity++ code, observe that there are 2^{26} special messages. Generalizing, it is easy to see that for a $(k + 2, k)$ Parity++ code, there are $2^{k-\log_2(k)-1}$ special messages.

Similar unequal message protection scheme can be used for providing DECTED protection to special messages, while non-special messages get SECDED protection. The code construction has been explained in detail in [30].

3.3 Error Detection and Correction

The received—possibly erroneous—vector $\mathbf{y}$ is divided into two parts, $\bar{\mathbf{c}}$ and η, with $\bar{\mathbf{c}}$ being the first $k+1$ bits of the codeword and η the additional non-linear redundancy bit ($\eta = 0$ for special messages and $\eta = 1$ for normal messages). There are three possible scenarios at the decoder: no (detectable) error, correctable error, or detected but uncorrectable error.

First, due to the Parity++ construction, every valid codeword has even weight. Thus, if $\bar{\mathbf{c}}$ has even weight, then the decoder concludes no error has occurred, i.e., $\bar{\mathbf{c}}$ was the original codeword. Second, if $\bar{\mathbf{c}}$ has odd weight and $\eta = 0$, the decoder attempts to correct the error. Since $\mathbf{G}_S$ is in systematic form, $\mathbf{H}_S$, its corresponding parity-check matrix can be easily retrieved. The decoder calculates the syndrome $\mathbf{s_1} = \mathbf{H}_S^T \bar{\mathbf{c}}$. If $\mathbf{s_1}$ is equal to a column in $\mathbf{H}_S$, then that corresponding bit in $\bar{\mathbf{c}}$ is flipped. Third, if $\bar{\mathbf{c}}$ has odd weight and either $\mathbf{s_1}$ does not correspond to any column in $\mathbf{H}_S$ or $\eta = 1$, then the decoder declares a DUE.

The decoding process described above guarantees that any single-bit error in a special codeword will be corrected, and any single-bit error in a normal codeword will be detected (even if the bit in error is η).

Let us take a look at two concrete examples for the $(10, 8)$ Parity++ code. Without any pre-mapping, a special message begins with $\log_2(3) + 1 = 4$ zeros. Let the original message be $\mathbf{m} = (00001011)$, which is encoded to $\mathbf{c} = (1011010110)$. Note that the first 4 bits of $\mathbf{c}$ is the systematic part of the special codeword. After passing through the channel, let the received vector be $\mathbf{y} = (1001010110)$, divided into $\bar{\mathbf{c}} = (1001010110)$ and $\eta = 0$. Since the weight of c is odd and $\eta = 0$, the decoder attempts to correct the error. The syndrome is equal to the 3rd column in $\mathbf{H}_S$, thus the decoder correctly flips the 3rd bit of $\bar{\mathbf{c}}$.

For the second example, let us begin with $\mathbf{m} = (11010011)$, which is encoded to (0011111101). After passing through the channel, the received vector is $\mathbf{y} = (0011011101)$. Since the weight of $\bar{\mathbf{c}}$ is odd and $\eta = 1$, the decoder declares a DUE. Note that for both normal and special codewords, if the only bit in error is η itself, then it is implicitly corrected since $\bar{\mathbf{c}}$ has even weight and will be correctly mapped back to $\mathbf{m}$ without any error detection or correction required.

3.4 Architecture

In an ECC protected cache, every time a cache access is initiated, the target block is sent through the ECC decoder/error detection engine. If no error is detected, the cache access is completed and the cache block is sent to the requester. If an error is detected, the block is sent through the ECC correction engine and the corrected block is eventually sent to the requester. Due to the protection mechanism, there is additional error detection/correction latency. Error detection latency is more critical than error correction as occurrence of an error is a rare event when compared to the processor cycle time and does not fall in the critical path. However, a block goes through the detection engine every time a cache access is initiated.

When using Parity++, the flow almost remains the same. Parity++ can detect all single-bit errors but has correction capability for "special messages." When a single-bit flip occurs on a message, the error detection engine first detects the error and stalls the pipeline. If the non-linear bit says it is a "special message" (non-linear bit is '0'), the received message goes through the Parity++ error correction engine which outputs the corrected message. This marks the completion of the cache access. If the non-linear bit says it is a non-special message (non-linear bit is "1"), it is checked if the cache line is clean. If so, the cache line is simply read back from the lower level cache or the memory and the cache access is completed. However, if the cache line is dirty and there are no other copies of that particular cache line, it leads to a crash or a roll back to checkpoint. Note that both Parity++ and SECDED have equal decoding latency of one cycle that is incurred during every read operation from an ECC protected cache. The encoding latency during write operation does not fall in the critical path and hence is not considered in the analyses.

The encoding energy overhead is almost similar for both Parity++ and SECDED. The decoding energy overheads are slightly different. For SECDED, the original message can be retrieved from the received codeword by simply truncating the additional ECC redundant bits. However, all received codewords need to be multiplied with the H-matrix to detect if any errors have occurred. For Parity++, all messages go through the chain of XOR gates for error detection and only the non-systematic non-special messages need to be multiplied with the decoder matrix to retrieve the original message. Since the error detection in Parity++ is much cheaper in terms of energy overhead than SECDED and the non-special messages only constitute about 20–25% of the total messages, the overall read energy in Parity++ turns out to be much lesser than SECDED.

3.5 Experimental Methodology

Parity++ was evaluated over applications from the SPEC 2006 benchmark suite. Two sets of core micro-architectural parameters (provided in Table 5) were chosen to understand the performance benefits in both a lightweight in-order (InO) processor and a larger out-of-order (OoO) core. Performance simulations were run using Gem5 [6], fast forwarding for one billion instructions and executing for two billion instructions.

The first processor was a lightweight single in-order core architecture with a 32kB L1 cache for instruction and 64kB L1 cache for data. Both the instruction and data caches were 4-way associative. The LLC was a unified 1MB 8-way associative L2 cache. The second processor was a dual core out-of-order architecture. The L1 instruction and data caches had the same configuration as the previous processor. The LLC comprises of both L2 and L3 caches. The L2 was a shared 512KB cache while the L3 was a shared 2MB 16-way associative cache. For both the baseline processors it was assumed that the LLCs (L2 for the InO processor and L2 and L3 for the OoO processor) have SECDED ECC protection.

Table 5 Core micro-architectural parameters

	Processor-1	Processor-2
Cores	1 (@ 2 GHz)	2 (@ 2 GHz)
Core type	InO (@ 2 GHz)	OoO (@ 2 GHz)
Cache line size	64B	64B
L1 Cache per core	32 KB I$, 64 KB D$	32 KB I$, 64 kB D$
L2 Cache	1 MB (unified) 8-way	512 KB (shared, unified) 8-way
L3 Cache	–	2 MB 16-way (shared)
Memory configuration	4 GB of 2133 MHz DDR3	8 GB of 2133 MHz DDR3
Nominal voltage	1 V	1 V

The performance evaluation was done only for cases where there are no errors. Thus, latency due to error detection was taken into consideration but not error correction as correction is rare when compared to the processor cycle time and does not fall in the critical path. In order to compare the performance of the systems with Parity++ against the baseline cases with SECDED ECC protection, the size of the LLCs was increased by ∼9% due to the lower storage overhead of Parity++ compared to SECDED. This is the iso-area case since the additional area coming from reduction in redundancy is used to increase the total capacity of the last level caches.

3.6 Results and Discussion

In this section the performance results obtained from the Gem5 simulations (as mentioned in Sect. 3.5) are discussed. Figures 8 and 9 show the comparative results for the two different sets of core micro-architectures across a variety of benchmarks from the SPEC2006 suite when using memory speculation. In both the evaluations, performance of the system with Parity++ was compared against that with SECDED.

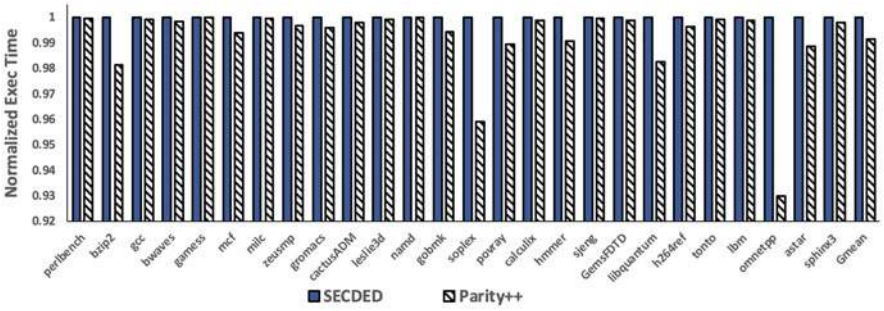


Fig. 8 Comparing normalized execution time of Processor-I with SECDED and Parity++

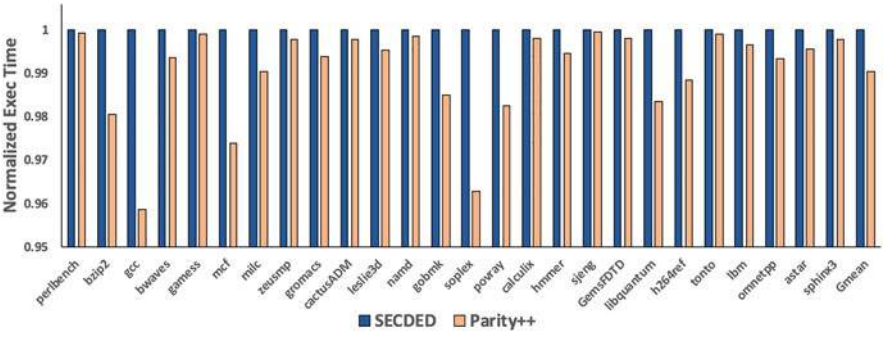


Fig. 9 Comparing normalized execution time of Processor-II with SECDED and Parity++

For both the core configurations, the observations are almost similar. It was considered that both Parity++ and SECDED protected caches have additional cache hit latency of one cycle (due to ECC decoding) for all read operations. The results show that with the exact same hit latency, Parity++ has up to 7% lower execution time than SECDED due to the additional memory capacity. The applications showing higher performance benefits are mostly memory intensive. Hence, additional cache capacity with Parity++ reduces overall cache miss rate. For most of these applications, this performance gap widens as the LLC size increases for Processor-II. The applications showing roughly similar performances on both the systems are the ones which already have a considerably lower LLC miss rate. As a result, increase in LLC capacity due to Parity++ does not lead to a significant improvement in performance.

On the other hand, if the cache capacity is kept constant (iso-capacity), Parity++ helps to save ∼5–9% of last level cache area (cache tag area taken into consideration) as compared to SECDED. Since the LLCs constitute more than 30% of the processor chip area, the cache area savings translate to a considerable amount of reduction in the chip size. This additional area benefit can either be utilized to make an overall smaller sized chip or it can be used to pack in more compute tiles to increase the overall performance of the system.

The results also imply that Parity++ can be used in SRAM based scratchpad memories used in embedded systems at the edge of the Internet-of-Things (IoT) where hardware design is driven by the need for low area, cost, and energy consumption. Since Parity++ helps in reducing area (in turn reducing SRAM leakage energy) and also has lower error detection energy [1], it provides a better protection mechanism than SECDED in such devices.

Acknowledgments The authors thank Dr. Mark Gottscho and Dr. Clayton Schoeny who were collaborators on the work presented in this chapter.

References

1. Alam, I., Schoeny, C., Dolecek, L., Gupta, P.: Parity++: lightweight error correction for last level caches. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), pp. 114–120 (2018). https://doi.org/10.1109/DSN-W.2018.00048
2. Alameldeen, A., Wood, D.: Frequent pattern compression: a significance-based compression scheme for L2 caches. Tech. rep., University of Wisconsin, Madison (2004)
3. Alameldeen, A.R., Wood, D.A.: Frequent pattern compression: a significance-based compression scheme for L2 caches (2004)
4. Bathen, L.A.D., Dutt, N.D.: E-RoC: embedded RAIDs-on-chip for low power distributed dynamically managed reliable memories. In: Design, Automation, and Test in Europe (DATE) (2011)
5. Bathen, L.A.D., Dutt, N.D., Nicolau, A., Gupta, P.: VaMV: variability-aware memory virtualization. In: Design, Automation, and Test in Europe (DATE) (2012)

6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). http://doi.acm.org/10.1145/2024716.2024718
7. Davydov, A., Tombak, L.: An alternative to the Hamming code in the class of SEC-DED codes in semiconductor memory. IEEE Trans. Inf. Theory **37**(3), 897–902 (1991)
8. Duwe, H., Jian, X., Kumar, R.: Correction prediction: reducing error correction latency for on-chip memories. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 463–475 (2015). https://doi.org/10.1109/HPCA.2015.7056055
9. Gelas, J.D.: The Intel Xeon E5 v4 review: testing broadwell-EP with demanding server workloads (2016). http://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review
10. Gottscho, M., Bathen, L.A.D., Dutt, N., Nicolau, A., Gupta, P.: ViPZonE: hardware power variability-aware memory management for energy savings. IEEE Trans. Comput. **64**(5), 1483–1496 (2015)
11. Gottscho, M., Schoeny, C., Dolecek, L., Gupta, P.: Software-defined error-correcting codes. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 276–282 (2016)
12. Gottscho, M., Schoeny, C., Dolecek, L., Gupta, P.: Software-defined ECC: heuristic recovery from uncorrectable memory errors. Tech. rep., University of California, Los Angeles, Oct 2017
13. Gottscho, M., Alam, I., Schoeny, C., Dolecek, L., Gupta, P.: Low-cost memory fault tolerance for IoT devices. ACM Trans. Embed. Comput. Syst. **16**(5s), 128:1–128:25 (2017)
14. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE International Workshop on Workload Characterization (IWWC) (2001)
15. Hamming, R.W.: Error detecting and error correcting codes. Bell Labs Tech. J. **29**(2), 147–160 (1950)
16. Hsiao, M.Y.: A class of optimal minimum odd-weight-column SEC-DED codes. IBM J. Res. Dev. **14**(4), 395–401 (1970)
17. Kaneda, S., Fujiwara, E.: Single byte error correcting – double byte error detecting codes for memory systems. IEEE Trans. Comput. **C-31**(7), 596–602 (1982)
18. Kim, J., Sullivan, M., Choukse, E., Erez, M.: Bit-plane compression: transforming data for better compression in many-core architectures. In: Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA) (2016)
19. Kim, J., Sullivan, M., Choukse, E., Erez, M.: Bit-plane compression: transforming data for better compression in many-core architectures. In: Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16, pp. 329–340. IEEE, Piscataway (2016). https://doi.org/10.1109/ISCA.2016.37
20. Lipasti, M.H., Wilkerson, C.B., Shen, J.P.: Value locality and load value prediction. In: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (1996)
21. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B.G.: Flikker: saving DRAM refresh-power through critical data partitioning. In: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2011)
22. Miguel, J.S., Badr, M., Jerger, N.E.: Load value approximation. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2014)
23. Miguel, J.S., Albericio, J., Jerger, N.E., Jaleel, A.: The Bunker Cache for spatio-value approximation. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2016)
24. Mittal, S., Vetter, J.: A survey of architectural approaches for data compression in cache and main memory systems. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1524–1536 (2015)
25. Nguyen, Q.: RISC-V tools (GNU toolchain, ISA simulator, tests) – git commit 816a252. https://github.com/riscv/riscv-tools

26. Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Base-delta-immediate compression: practical data compression for on-chip caches. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT) (2012)
27. Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Base-delta-immediate compression: practical data compression for on-chip caches. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pp. 377–388. ACM, New York (2012). http://doi.acm.org/10.1145/2370816.2370870
28. Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2013)
29. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: approximate data types for safe and general low-power computation. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) (2011)
30. Schoeny, C., Sala, F., Gottscho, M., Alam, I., Gupta, P., Dolecek, L.: Context-aware resiliency: unequal message protection for random-access memories. In: Proc. IEEE Information Theory Workshop, Kaohsiung, pp. 166–170, Nov 2017
31. Song, J., Bloom, G., Palmer, G.: SuperGlue: IDL-based, system-level fault tolerance for embedded systems. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2016)
32. Waterman, A.: RISC-V proxy kernel – git commit 85ae17a. https://github.com/riscv/riscv-pk/commit/85ae17a
33. Waterman, A., Lee, Y.: Spike, a RISC-V ISA Simulator – git commit 3bfc00e. https://github.com/riscv/riscv-isa-sim
34. Waterman, A., Lee, Y., Patterson, D., Asanovic, K.: The RISC-V instruction set manual volume I: user-level ISA version 2.0 (2014). https://riscv.org
35. Yang, J., Zhang, Y., Gupta, R.: Frequent value compression in data caches. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2000)
36. Yazdanbakhsh, A., Pekhimenko, G., Thwaites, B., Esmaeilzadeh, H., Mutlu, O., Mowry, T.C.: Mitigating the memory bottleneck with approximate load value prediction. IEEE Des. Test **33**(1), 32–42 (2016)
37. Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., Lotfi-Kamran, P.: AxBench: a multiplatform benchmark suite for approximate computing. IEEE Des. Test **34**(2), 60–68 (2017)

Resource Management for Improving Overall Reliability of Multi-Processor Systems-on-Chip

Yue Ma, Junlong Zhou, Thidapat Chantem, Robert P. Dick, and X. Sharon Hu

1 Introduction

This section presents the concepts and models associated with soft-error reliability and lifetime reliability, and reviews the related work on these topics.

1.1 Background

Modern multi-processor systems on a chip (MPSoCs) may contain both multicore processors and integrated GPUs, which are especially suitable for real-time embedded applications requiring massively parallel processing capabilities. Since MPSoCs

Y. Ma · X. S. Hu (✉)
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA
e-mail: yma1@nd.edu; shu@nd.edu

J. Zhou
School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China
e-mail: jlzhou@njust.edu.cn

T. Chantem
Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Arlington, VA, USA
e-mail: tchantem@vt.edu

R. P. Dick
Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA
e-mail: dickrp@umich.edu

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_10

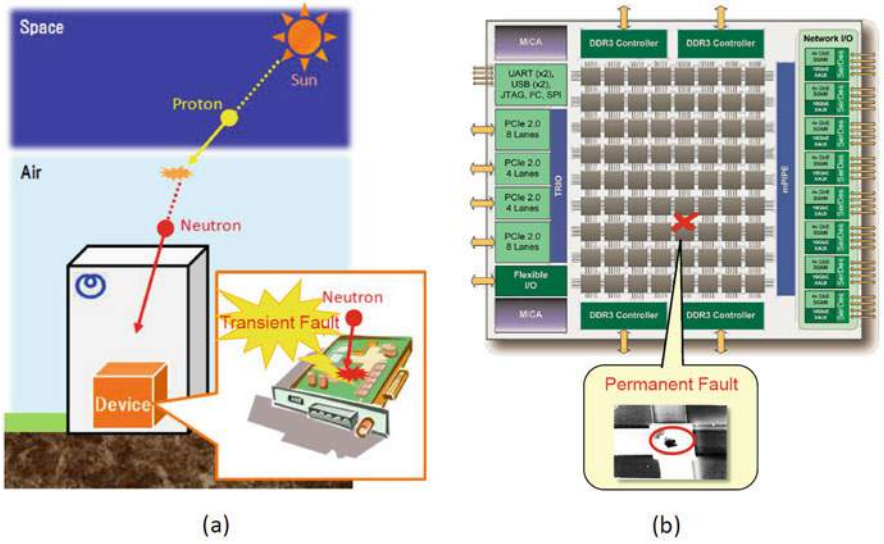


Fig. 1 Illustration of transient and permanent faults

offer good performance and power consumption, they have been widely used in many real-time applications such as consumer electronics, automotive electronics, industrial automation, and avionics [1]. For these applications, the MPSoC needs to satisfy deadline, quality-of-service (e.g., resolution of video playback), and reliability requirements. The reliability requirements include both soft-error reliability (SER), influenced by transient faults, and lifetime reliability (LTR), influenced by permanent faults. This chapter presents approaches to improving SER and/or LTR while satisfying deadline and quality-of-service requirements for real-time embedded systems.

Transient faults are mainly caused by high-energy particle strikes, e.g., resulting from spallation from cosmic rays striking atoms in the upper atmosphere [2] (see Fig. 1a). Transient faults may lead to errors that appear for a short time and then disappear without damaging the device or shortening its lifetime; these are called soft errors. They may prevent tasks from completing successfully. SER is used to quantify the probability that tasks will complete successfully without errors due to transient faults. SER can be increased by using reliability-aware techniques such as replication, rollback recovery, and frequency elevation, which either tolerate transient faults or decrease their rates.

Permanent faults are caused by wear in integrated circuits. An example is illustrated in Fig. 1b. Permanent faults can lead to errors that persist until the faulty hardware is repaired or replaced. Multiple wear-out effects such as electromigration (EM), stress migration (SM), time-dependent dielectric breakdown (TDDB), and thermal cycling (TC) can lead to permanent faults. The rates of these effect depend exponentially on temperature. In addition, thermal cycling depends on the

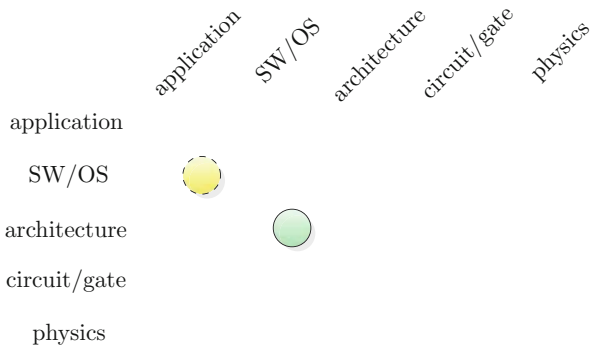


Fig. 2 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

temperature range, maximum temperature, and cycle frequency. To improve LTR, temperature peaks and variation must be limited.

To reduce the cost of repairing/replacing an MPSoC system and maintain some desired level of quality-of-service, improving SER due to transient faults and LTR due to permanent faults become an imperative design concern. In this chapter we present two techniques that optimize SER and LTR separately and show how to make appropriate trade-offs between them for improving overall system reliability. Figure 2 illustrates the abstraction layers representing the main contribution of this chapter.

1.2 Related Work

Considerable research has been done on improving SER. Haque et al. [3] present an energy-efficient task replication method to achieve a high SER target for periodic real-time applications running on a multicore system with minimum energy consumption. Salehi et al. [4] propose a low-overhead checkpointing-based rollback recovery scheme to increase system SER and reduce the number of checkpoints for fault-tolerant real-time systems. Zhou et al. [5] improve system SER by judiciously determining proper replication and speedup of tasks. Zhou and Wei [6] describe a stochastic fault-tolerant task scheduling algorithm that specifically considers uncertainty in task execution caused by transient fault occurrences to increase SER under task deadline constraints. These work increase SER but do not consider permanent faults.

Many studies have focused on increasing LTR. Huang et al. [7] describe an analytical model to derive the LTR of multicore systems and a simulated annealing algorithm to reduce core temperature and temperature variation to improve system LTR. Chantem et al. [8] present a dynamic task assignment and scheduling scheme to maximize system LTR by mitigating core wear due to thermal cycling. Ma et

al. [1] optimize system LTR by establishing an online framework that dynamically controls cores' utilization. Das et al. [9, 10] improve the LTR of network-on-chips (NoCs) and also solve the energy–reliability trade-off problem for multimedia MPSoCs. However, these approaches neglect transient faults.

There is research on handling SER and LTR together. Zhou et al. [5] propose a task frequency and replication selection strategy that balances SER and LTR to maximize system availability. Ma et al. [11] establish an online framework for increasing SER and LTR of real-time systems running on "big–little" type MPSoCs. A genetic algorithm based approach [12] that determines task mappings and frequencies is developed to jointly improve SER and LTR. Aliee et al. [13] adopt mean time to failure (MTTF) as the common metric to evaluate SER and LTR and design a success tree based scheme for reliability analysis for embedded systems. Unlike work [5, 11–13] that ignore the variations in performance, power consumption, and reliability parameters, Gupta et al. [14] explore the possibility of constructing reliable systems to compensate for the variability effects in hardware through software controls. These efforts consider CPU reliability but ignore the reliability effects of GPUs.

1.3 Soft-Error Reliability Model

SER is the probability that no soft errors occur in a particular time interval [5], i.e.,

$$r = e^{-\lambda(f)\times U\times|\Delta t|}, \tag{1}$$

where f is the core frequency, $|\Delta t|$ is the length of the time interval, U is the core's utilization within $|\Delta t|$, and $\lambda(f)$ is the average fault rate depending on f [5]. Specifically, we have

$$\lambda(f) = \lambda_0 \times 10^{\frac{d(f_{\max}-f)}{f_{\max}-f_{\min}}}, \tag{2}$$

where λ_0 is the average fault rate at the maximum core frequency. $f_{\min}$ and $f_{\max}$ are the minimum and maximum core frequency, and d ($d > 0$) is a hardware-specific constant indicating the sensitivity of fault rate to frequency scaling. Reducing frequency leads to an exponential increase in fault rate because frequency is a roughly linear function of supply voltage. As frequency reduces, supply voltage decreases, decreasing the critical charge (i.e., the minimum amount of charge that must be collected by a circuit to change its state) and exponentially increasing fault rate [15].

Since CPU and GPU fabrication processes are similar, the device-level SER model above applies to both. Let r_G and r_i ($i = 1, 2, \ldots, m$) represent the SER of the GPU and the ith CPU core, respectively. As the correct operation of an MPSoC system-level depends on the successful execution of GPU and CPU cores, the system-level SER is calculated as the product of reliabilities of all individual cores, i.e.,

$$R = r_G \times \prod_{i=1}^{m} r_i. \tag{3}$$

1.4 Lifetime Reliability Model

MTTF is commonly used to quantify LTR. We focus on four main failure mechanisms: EM, TDDB, SM, and TC. EM refers to the dislocation of metal atoms caused by momentum imparted by electrical current in wires and vias [16]. TDDB refers to the deterioration of the gate oxide layer [17]. SM is caused by the directionally biased motion of atoms in metal wires due to mechanical stress caused by thermal mismatch between metal and dielectric materials [18]. TC is wear due to thermal stress induced by mismatched coefficients of thermal expansion for adjacent material layers [19].

The system-level MTTF modeling tool introduced by Xiang et al. [20] can be used to estimate LTR when considering the above four failure mechanisms. This tool integrates three levels of models, i.e., device-, component-, and system-level models. At the device level, wear due to the above four mechanisms is modeled. The modeling tool accounts for the effect of using multiple devices in a component upon fault distributions, e.g., the effects of EM are most appropriately modeled using a lognormal distribution at the device level, but with a Weibull distribution for components containing many devices. Based on the device-level reliability models and temporal failure distributions, component-level MTTF is calculated [20]. Then, based on component-level reliability, the system-level MTTF is obtained by Monte Carlo simulation.

2 LTR and SER Optimization

This section introduces two approaches for LTR and SER optimization, and discusses the trade-off between them.

2.1 LTR Optimization

EM, SM, and TDDB wear rates depend exponentially on temperature. However, wear due to thermal cycling depends on the amplitude (i.e., the difference between the proximal peak and valley temperature), period, and maximum temperature of thermal cycles. Figure 3 summarizes some system MTTF data obtained from the system-level LTR modeling tool with default settings [20]. Figure 3a–c depicts the MTTF of an example system as a function of the amplitude, period, and peak

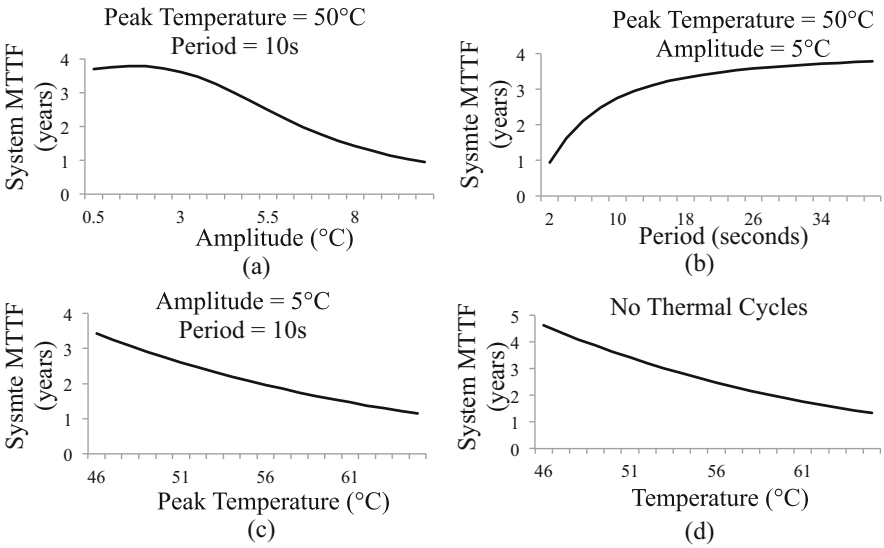


Fig. 3 System MTTF due to: (**a**) amplitude of thermal cycle; (**b**) period of thermal cycles; (**c**) peak temperature of thermal cycles; and (**d**) temperature without thermal cycles

temperature of thermal cycles, respectively. As a comparison, Fig. 3d shows the system MTTF due to temperature alone without thermal cycles. As can be seen from Fig. 3, system MTTF generally increases for lower temperatures and smaller thermal cycles.

A system's LTR is determined by its operating temperature and thermal cycles. Given that lower frequencies and voltages lead to higher utilization but lower temperatures, one method to improve system MTTF is to control core utilization. For example, we have developed a framework called Reliability-Aware Utilization Control (RUC) [21] to mitigate the effects of both operating temperature and thermal cycling. RUC consists of two controllers. The first controller reduces the peak temperature by periodically reducing core frequencies subject to task deadline requirements. Although frequent changes in core frequency helps to reduce peak temperature, they may increase the frequency of thermal cycling and reduce lifetime reliability. Hence, the second controller minimizes thermal cycling wear by dynamically adjusting the period of the first controller to achieve longer thermal cycles as well as lower peak temperature.

2.2 SER Optimization

Recovery allocation strategies and task execution orders can affect system-level soft-error reliability (as shown in Fig. 4). In this example, there are four tasks that

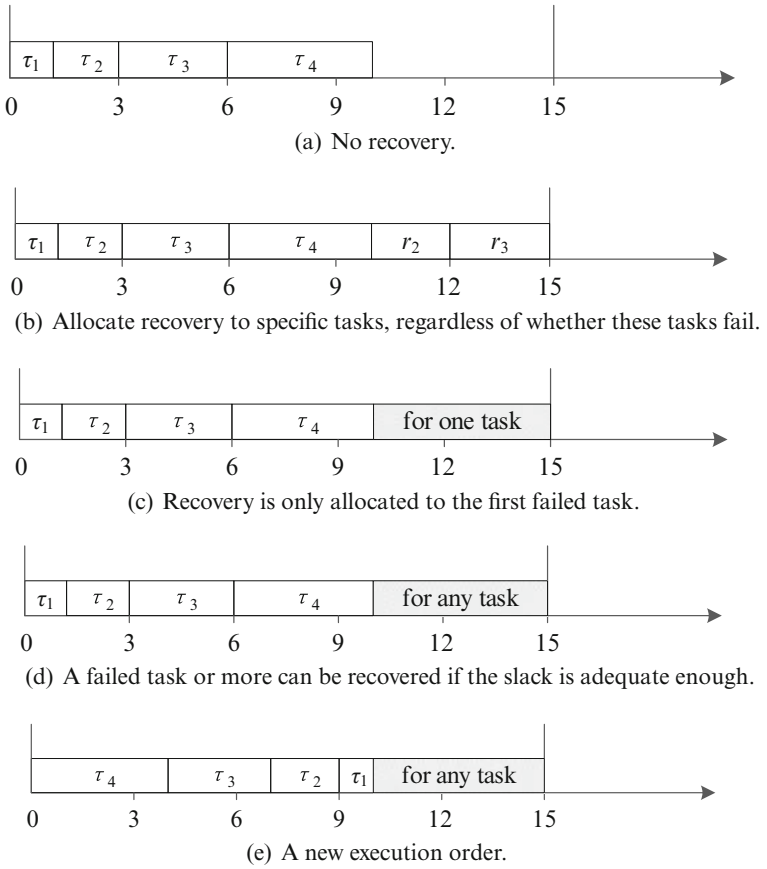


Fig. 4 Motivating examples illustrate different recovery allocation strategies and task execution order affect system-level SER. (**a**) No recovery. (**b**) Statically allocate recovery to specific tasks, regardless of whether these tasks fail. (**c**) Recovery is only allocated to the first failed task. (**d**) A failed task or more can be recovered if the slack is adequate. (**e**) A new task execution order

share a common period of 15 s. We further suppose the worst-case execution times of the tasks are 1, 2, 3, and 4 s. All tasks in the set execute at the highest core frequency. As indicated by the reliability model presented in Sect. 1, the SERs of the tasks are 0.904, 0.819, 0.741, and 0.670.

If no recovery is allowed as shown in Fig. 4a, the system-level SER, i.e., the probability that all tasks can complete successfully, is 0.368. Allowing recovery of some tasks increases SER. One method is to allocate recoveries to tasks offline [5]. Figure 4b represents a better solution for maximizing the system-level SER, in which tasks τ_2 and τ_3 have recoveries r_2 and r_3. In this case, the system-level SER is 0.547. Another approach allocates recovery online [22]. Figure 4c shows a scenario where the first failed task has a recovery [22]. The system-level SER is 0.686, which

is higher than that in Fig. 4b. However, although the slack is dynamically used in Fig. 4c, only one task can be recovered.

In the above online recovery allocation example, a failed task is recovered if the remaining slack is adequate, and tasks consume slack on a first-come, first-served basis (see Fig. 4d). For example, task τ_2 can recover even if τ_1 fails. However, task τ_3 cannot recover if both τ_1 and τ_2 fail because the remaining slack for τ_3 is only 2 s. Task τ_4 can recover only when all tasks succeed or only τ_1 fails. Hence, the probabilities of recovering τ_3 and τ_4 are 0.983 and 0.607, and the system-level SER is 0.716. Now, consider the impact of task scheduling on the system-level SER. Figure 4e represents a new schedule where the task's priority is the inverse of its execution time. In this case, the probabilities of recovering τ_1, τ_2, τ_3, and τ_4 are 0.792, 0.670, 0.670, and 1.000. In contrast with Fig. 4d, the task with the lowest SER, τ_4, can always be recovered, but the system-level SER is 0.692. Hence, a scheduling algorithm that simply improves the probability of recovery for some specific tasks may not be a good solution.

Based on these observations, we design an SER improvement framework [23] that statically schedules tasks and dynamically allocates recoveries. The framework is composed of a simple and fast scheduling algorithm for special task sets and a powerful scheduling algorithm for general task sets. For more details of the two scheduling algorithms, readers can refer to [23].

3 Trade-Off Between LTR and SER

Certain design decisions (e.g., task mapping and voltage scaling) may increase LTR but decrease SER, and vice versa. In other words, improving overall reliability requires trade-offs between LTR and SER. Recently, several efforts have focused on these trade-offs. Below, we describe two case studies in LTR and SER trade-off: (1) "big–little" type MPSoCs and (2) CPU–GPU integrated MPSoCs.

3.1 "Big–Little" MPSoCs

To address power/energy concerns, various heterogeneous MPSoCs have been introduced. A popular MPSoC architecture often used in power/energy-conscious real-time embedded applications is composed of pairs of high-performance (HP) cores and low-power (LP) cores. Such HP and LP cores present unique performance, power/energy, and reliability trade-offs. Following the terminology introduced by ARM, we refer to this as the "big–little" architecture. Nvidia's variable symmetric multiprocessing architecture is such an example [24].

Executing tasks on an LP core improves LTR by reducing temperature and improves SER through a higher core frequency. Although the primary goal of "big–little" MPSoCs is to reduce power consumption by executing a light workload on

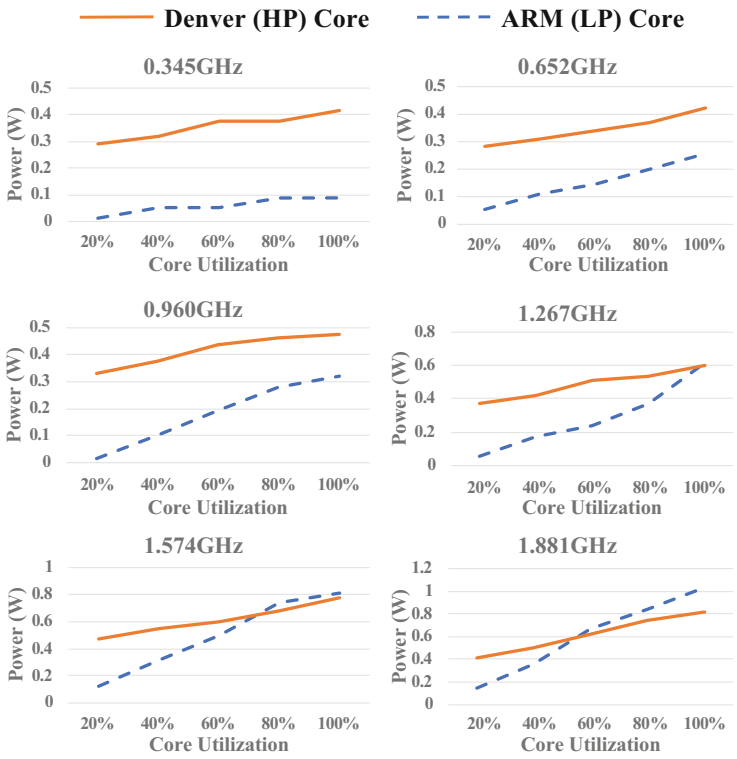


Fig. 5 The measured power consumptions of HP (Denver) core and LP (ARM) core on Nvidia's TX2 chip as functions of utilization and frequency

the LP cores, there are circumstances in which an LP core consumes more power than an HP core. Carefully characterizing the power consumption behavior of HP and LP cores is necessary. For example, the power consumption of the HP core and LP core on Nvidia's TX2[1] is shown in Fig. 5. The LP core consumes less power than the HP core only when the core frequency is low and the workload is light. One possible reason for this phenomenon is that the HP and LP cores have different microarchitectures, as is the case with the TX2. Another possible reason is that the transistors in the HP core and LP core have different threshold voltages. The LP core has low leakage power but requires high voltage to operate at higher frequencies. On the other hand, the HP core can work at high frequency with a low voltage.

The above observations reveal that in order to reduce power consumption of MPSoCs and improve reliability, it is necessary to fully account for the power

[1]Note that TX2 is composed of ARM Cortex A57 cores that support multithreading, and Nvidia' Denver cores for high single-thread performance with dynamic code optimization. Denver cores can be treated as HP cores and ARM cores can be treated as LP cores when running single-threaded applications.

features of heterogeneous cores and carefully map tasks to the most appropriate cores. A good guideline is to run tasks having short execution times on LP cores with low frequencies and tasks having long execution times on HP cores with high frequencies. The execution models of HP and LP cores must also be considered. For example, HP and LP cores on Nvidia's TK1 cannot execute at the same time. However, on Nvidia's TX2, HP and LP cores can work simultaneously. Although an HP core and an LP core can execute at different frequencies, all HP cores must share one frequency, as must LP cores. Hence, a strategy to improve reliability should migrate tasks dynamically and consider both the power features and execution models of HP and LP cores. Using this guideline, we have developed frameworks for different hardware platforms to improve soft-error reliability under lifetime reliability, power consumption, and deadline constraints [1, 11].

3.2 CPU–GPU Integrated MPSoCs

Thanks to the massively parallel computing capability offered by GPUs and the general computing capability of CPUs, MPSoCs with integrated GPUs and CPUs have been widely used in many soft real-time embedded applications, including mobile devices [25] and intelligent video analytics. For many such applications, SER due to transient faults and LTR due to permanent faults are major design concerns. A common reliability improvement objective is to maximize SER under an LTR constraint.

An application task set is used to illustrate how a task's execution time depends on whether it executes on the same core as the operating system. The varying execution times of tasks change the overall workload and operating temperature, influencing LTR and SER. Experiments were performed on Nvidia's TK1 chip (with CUDA 6.5) with default settings to measure task execution times. Six tasks from different benchmark suites were executed (see Table 1). Each task's increase in CPU time resulting from executing on a different core than the operating system is shown in Fig. 6 and the averages of additional GPU times are shown in Table 2. For all tasks, the additional CPU times can be significant and are input dependent. In contrast to the additional CPU time, the additional GPU time is negligible: the additional GPU times of all measured application tasks are less than 1% of the tasks' execution times. This increase can be ignored in most soft real-time applications. Similar phenomena can be observed for other platforms. On Nvidia's TX2 chip, the additional CPU times of application tasks are illustrated in Fig. 7.

The above observations imply that a task's CPU time increases if executed on a different core than the operating system, but its GPU time does not change. Since both LTR and SER increase with a lighter workloads, this observation reveals that we should consider what resources tasks use when assigning them to cores. Generally, the primary core, on which the operating system runs, should be reserved for application tasks that require GPU resources to complete.

Table 1 Application tasks used to measure additional execution times

Name	Description	Source
VectorAdd	Vector addition	CUDA samples [26]
SimpleTexture	Texture use	
MatrixMul	Matrix multiplication	
Gaussian	Gaussian elimination	Rodinia [27]
BFS	Breadth-first search	
Backprop	Back propagation	

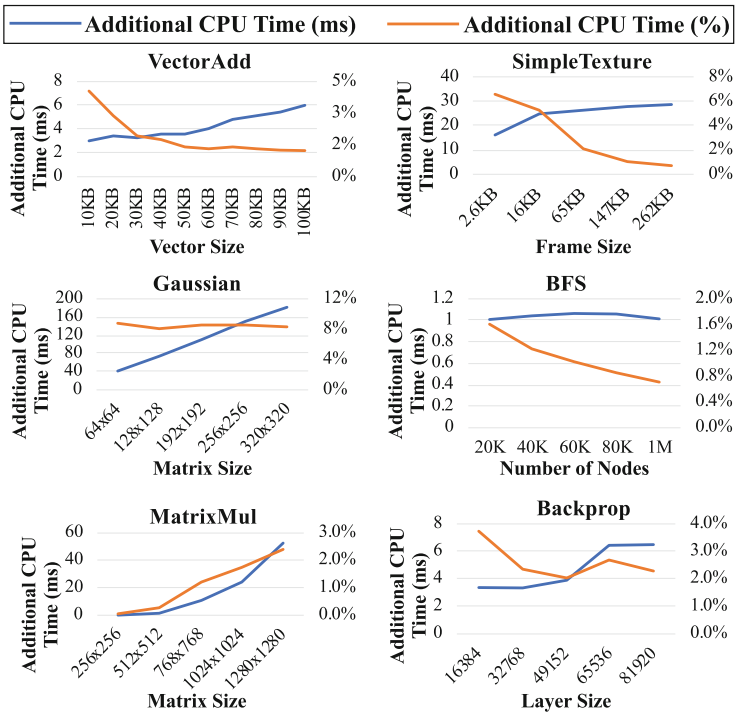


Fig. 6 Measured additional CPU times on TK1 for tasks executing on non-OS CPU cores

Table 2 Observed additional GPU time on TK1

	Additional GPU time	
Tasks	(ms)	(%)
VectorAdd	0.38	0.01
SimpleTexture	0.09	0.00
MatrixMul	0.22	0.11
Gaussian	0.38	0.00
BFS	0.003	0.20
Backprop	0.31	0.68

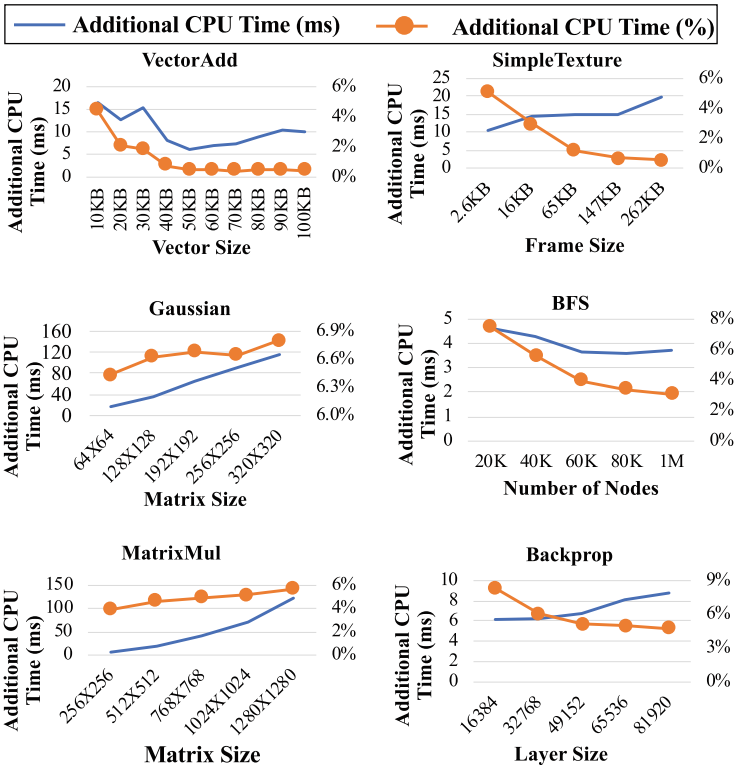


Fig. 7 Measured additional CPU times on TX2 for tasks executing on CPU cores that are different from the core where the operating system runs

4 Conclusion

Real-time embedded system soft-error and lifetime reliabilities are important. Generally, increasing a core's frequency, allocating recoveries and allowing replications improve soft-error reliability, but may increase operating temperature thereby reducing lifetime reliability. MPSoCs used in many applications are heterogeneous and integrate high-performance cores, low-power cores, and even GPUs. System designers should model the task-dependent power consumptions and execution times of the cores available to them, and use these models to solve the SER and LTR trade-off problem.

References

1. Ma, Y., Chantem, T., Dick, R.P., Wang, S., Hu, X.S.: An on-line framework for improving reliability of real-time systems on "big-little" type MPSoCs. In: Proceedings of Design, Automation and Test in Europe, March, pp. 1–6 (2017)
2. Henkel, J., et al.: Design and architectures for dependable embedded systems. In: Proceedings of the International Conference Hardware/Software Codesign and System Synthesis, October, pp. 69–78 (2011)
3. Haque, M.A., Aydin, H., Zhu, D.: On reliability management of energy-aware real-time systems through task replication. IEEE Trans. Parallel Distrib. Syst. **28**(3), 813–825 (2017)
4. Salehi, M., Tavana, M.K., Rehman, S., Shafique, M., Ejlali, A., Henkel, J.: Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. IEEE Trans. VLSI Syst. **24**(7), 2426–2437 (2016)
5. Zhou, J., Hu, X.S., Ma, Y., Wei, T.: Balancing lifetime and soft-error reliability to improve system availability. In: Proceedings of Asia and South Pacific Design Automation Conference, January, pp. 685–690 (2016)
6. Zhou, J., Wei, T.: Stochastic thermal-aware real-time task scheduling with considerations of soft errors. J. Syst. Softw. **102**, 123–133 (2015)
7. Huang, L., Yuan, F., Xu, Q.: On task allocation and scheduling for lifetime extension of platform-based MPSoC designs. IEEE Trans. Parallel Distrib. Syst. **22**(12), 789–800 (2011)
8. Chantem, T., Xiang, Y., Hu, X.S., Dick, R.P.: Enhancing multicore reliability through wear compensation in online assignment and scheduling. In: Proceedings of Design, Automation and Test in Europe, March, pp. 1373–1378 (2013)
9. Das, A., Kumar, A., Veeravalli, B.: Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In: Proceedings of Design, Automation and Test in Europe, March, pp. 689–694 (2013)
10. Das, A., Kumar, A., Veeravalli, B.: Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs. In: Proceedings of Design, Automation and Test in Europe, March, pp. 1–6 (2014)
11. Ma, Y., Zhou, J., Chantem, T., Dick, R.P., Wang, S., Hu, X.S.: On-line resource management for improving reliability of real-time systems on "big-little" type MPSoCs. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **39**, 88–100 (2018)
12. Das, A., Kumar, A., Veeravalli, B., Bolchini, C., Miele, A.: Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In: Proceedings of Design, Automation and Test in Europe, March, pp. 1–6 (2014)
13. Aliee, H., Glaß, M., Reimann, F., Teich, J.: Automatic success tree-based reliability analysis for the consideration of transient and permanent faults. In: Proceedings of Design, Automation and Test in Europe, March, pp. 1621–1626 (2013)
14. Gupta, P., et al.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **32**(1), 8–23 (2012)
15. Harucha, P., Suensson. C.: Impact of CMOS technology scaling on the atmospheric neutron soft error rate. IEEE Trans. Nucl. Sci. **47**(6), 2586–2594 (2000)
16. Srinivasan, J., Adve, S., Bose, P., Rivers, J.: The impact of technology scaling on lifetime reliability. In: Proceedings of International Conference Dependable Systems and Networks, June, pp. 177–186 (2004)
17. Srinivasan, J., Adve, S., Bose, P., Rivers, J.: Exploiting structural duplication for lifetime reliability enhancement. In: Proceedings of International Symposium on Computer Architecture, June, pp. 520–531 (2005)
18. JEDEC Solid State Technology Association: Failure mechanisms and models for semiconductor devices. Joint Electron Device Engineering Council. Technical Report, August 2003, JEP 122-B
19. Ciappa, M., Carbognani, F., Fichtner, W.: Temperature-aware microarchitecture: modeling and implementation. IEEE Trans. Device Mater. Reliab. **3**(4), 191–196 (2003)

20. Xiang, Y., Chantem, T., Dick, R.P., Hu, X.S., Shang, L.: System-level reliability modeling for MPSoCs. In: Proceedings of International Conference Hardware/Software Codesign and System Synthesis, October, pp. 297–306 (2010)
21. Ma, Y., Chantem, T., Dick, R.P., Hu, X.S.: Improving system-level lifetime reliability of multicore soft real-time systems. IEEE Trans. VLSI Syst. **25**(6), 1895–1905 (2017)
22. Zhao, B., Aydin, H., Zhu, D.: Enhanced reliability-aware power management through shared recovery technology. In: Proceedings of International Conference Computer-Aided Design, November, pp. 63–70 (2009)
23. Ma, Y., Chantem, T., Dick, R.P., Hu, X.S.: Improving reliability for real-time systems through dynamic recovery. Proceedings of Design, Automation and Test in Europe, March, pp. 515–520 (2018)
24. Nvidia: Variable SMP (4-plus-1) a multi-core CPU architecture for low power and high performance. https://www.nvidia.com/content/PDF/tegra white papers (2018). Online. Accessed Oct 2018
25. Samsung: Samsung Exynos. https://www.samsung.com/semiconductor/minisite/exynos/ (2018). Online. Accessed Oct 2018
26. Nvidia: CUDA samples. http://docs.nvidia.com/cuda/cuda-samples/index.html (2018). Online. Accessed Oct 2018
27. University of Virginia: Rodinia: accelerating compute-intensive applications with accelerators. https://rodinia.cs.virginia.edu (2018). Online. Accessed Oct 2018

Part III
Cross-Layer Resilience: Bridging the Gap Between Circuit and Architectural Layer

Daniel Mueller-Gritschneder

Today's design teams, as their forerunners in the past, struggle to master the ever-increasing complexity in chip design driven by new applications such as autonomous driving, complex robotics or embedded machine learning, which demand higher performance under strict power, area, and energy constraints. While Moore's law was providing improvements on all these objectives regularly by moving to the next technology node, a slow-down in scaling is observed nowadays. Yet, design teams came up with intelligent new design principles to provide further performance gains, most famously multi-core and many-core CPUs as well as GPUs combined with complex memory organizations with several level of hierarchy. Additionally, new computing principles moved into the focus of research and industry including near-threshold computing (NTC) for ultra-low-energy operations or the use of runtime-reconfigurable architectures based on field-programmable gate arrays (FPGAs), which flexibly provide specialized compute kernels to boost performance at low power costs.

One major design challenge in these new computing platforms is dependability whenever high system availability is demanded (always on) or, even more strict, in safety-critical applications. Dependable computing requires resilience against a whole range of error sources such as radiation-induced soft errors, aging effects, e.g., caused by Bias Temperature Instability (BTI) or Hot Carrier Injection (HCI), harsh environmental conditions, process variations or supply voltage noise. Depending on the chosen computing principle, new dependability challenges arise, e.g., configuration bits need to be made resilient against errors in FPGAs while NTC-based systems are more sensitive to process variations.

Resilience can be achieved at different layers of the design stack (SW, Compiler, Architecture, Circuit, Device). Traditionally, different parts of the design team look at different layers individually. Hence, resilience can lead to high design overheads

D. Mueller-Gritschneder (✉)
Technical University of Munich, Munich, Germany
e-mail: daniel.mueller@tum.de

as counter-measures on every layer are stacked on top of each other. This leads to the idea of cross-layer resilience. This design method looks for a resilience scheme, in which protection mechanisms work in a cooperative fashion across different layers of the design stack in order to reduce overheads, and, hence, cost. For hardware design, this cross-layer approach can be applied successfully across the architectural and circuit layer. This is demonstrated within the following three chapters.

The chapter titled *Cross-Layer Resilience against Soft Errors: Key Insights* focuses on the analysis and cross-layer protection against soft errors for different system components ranging from embedded processors to SRAM memories and accelerators. While Moore's law is driven mainly by high performance systems, many safety-critical systems are still designed at much older technology nodes to avoid reliability challenges. Yet, due to rising demand of processing power, e.g., for autonomous driving, newer technology nodes become mandatory. This makes it more costly to assure protection against soft errors as their chance of occurrence increases. Soft errors need to be detected and handled in any safety-critical application because they may cause malfunction of the system due to corruption of data or flow of control. Systems deploy protection techniques such as hardening and redundancy at different layers of the system stack (circuit, logic, architecture, OS/schedule, compiler, software, algorithm). Here, cross-layer resilience techniques aim at finding lower cost solutions by providing accurate estimation of soft error resilience combined with a systematic exploration of protection techniques that work collaboratively across the system stack. This chapter provides key insights on applying the cross-layer resilience principle in a lessons-learned fashion.

The chapter *Online Test Strategies and Optimizations for Reliable Reconfigurable Architectures* discusses cross-layer dependability of runtime-reconfigurable architectures based on FPGAs. Such FPGAs are often using the newest technology nodes. Hence, resilience is a major concern as newer nodes experience aging effects earlier and may suffer from higher susceptibility to environmental stress. Device aging can lead to malfunction of the system before its end-of-life, and hence, is a major dependability concern. Incorrect functionality can be detected by executing online built-in self-tests regularly on the device. Two orthogonal online tests are presented in this chapter. These tests can ensure the correctness of the configuration bits of the reconfigurable fabric as well as of the functional parts. Additionally, a design method called module diversification is presented, which enables to recover from faults by providing a self-repair feature. Finally, a design method is presented that implements a stress-aware FPGA placement method. It allows to slow down system degradation due to aging effects and prolongs system lifetime.

The final chapter *Reliability Analysis and Mitigation of Near-Threshold Voltage (NTC) Caches* targets NTC low-energy design and the related reliability concerns. This includes impact of soft error, aging, and process variation while operating at near-threshold voltage with special focus on on-chip caches. The idea is to save energy by scaling supply voltage of the cache blocks. The presented methods guide the designer to optimized NTC cache organizations using a cross-layer reliability analysis approach covering 6T and 8T SRAM cells. Overall, the three chapters bridge the architecture and circuit layer gap while also expanding to adjacent layers of the design stack.

Cross-Layer Resilience Against Soft Errors: Key Insights

Daniel Mueller-Gritschneder, Eric Cheng, Uzair Sharif, Veit Kleeberger, Pradip Bose, Subhasish Mitra, and Ulf Schlichtmann

1 Introduction

Two tasks need to be solved when designing systems for safety-critical application domains: firstly, the safety of the intended functionality (SoiF) must be guaranteed. SoiF focuses on the ability of the system to sense its environment and act safely. Achieving SoiF becomes a highly challenging task due to the rising complexity of various safety-critical applications such as autonomous driving or close robot–human interaction, which may require complex sensor data processing and interpretation. Secondly, and no less important, the system must also always remain or transit into a safe state given the occurrence of random hardware faults. To achieve this requirement, the system must be capable of detecting as well as handling or correcting possible errors. Safety standards such as ISO26262 for road vehicles define thresholds on detection rates for different automotive safety integration levels (ASIL) depending on the severity of a possible system failure, the controllability by the driver, and the nominal usage time of the system. It is commonly understood that safety-critical systems must be designed from the beginning with the required error protection in mind [39] and that for general-purpose computing systems, error protection is required to achieve dependable computing [19, 21].

U. Sharif · D. Mueller-Gritschneder (✉) · U. Schlichtmann · V. Kleeberger
Infineon Technologies AG, Neubiberg, Germany
e-mail: daniel.mueller@tum.de; uzair.sharif@tum.de; ulf.schlichtmann@tum.de

E. Cheng · S. Mitra
Stanford University, Stanford, CA, USA
e-mail: eccheng@stanford.edu; subh@stanford.edu

P. Bose
IBM, New York, NY, USA
e-mail: pbose@us.ibm.com

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_11

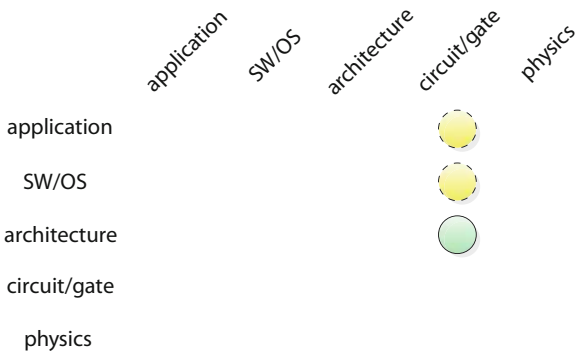


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

This requirement is becoming increasingly challenging as integrated systems—following the continuous trend of Dennard scaling—become more susceptible to fault sources due to smaller transistor dimensions and lower supply voltages. As transistor dimensions scale down the charge stored in memory cells such as SRAM or flip-flops decreases. Soft errors occur due to charge transfers when primary or secondary particles from cosmic radiation hit the silicon [11]. This charge transfer may lead to the corruption of the value stored in the cell. This is referred to as a "soft error" as it does not permanently damage the cell. The vulnerability of cells increases even further with shrinking supply voltage levels or sub-threshold operation. Thus, for the design of safety-critical digital systems, the protection against radiation-induced soft errors is a crucial factor to avoid unacceptable risks to life or property.

This reality motivates methods that aim to increase the resilience of safety-critical systems against radiation-induced soft errors in digital hardware. Common protection techniques against soft errors either harden the memory elements to reduce the probability of soft errors occurring or add redundancy at different layers of the design (circuit, logic, architecture, OS/schedule, compiler, software, algorithm) to detect data corruptions, which can subsequently be handled or corrected by appropriate error handlers or recovery methods. Each protection technique adds overheads and, hence, additional costs. Especially, adding protection techniques on top of each other at all layers—not considering combined effects—may lead to inefficient protection and non-required redundancy. The idea of cross-layer resiliency is to systematically combine protection techniques that work collaboratively across the layers of the system stack. The target is to find more efficient protection schemes with the same soft error resilience at a lower cost than can be reached by ignoring cross-layer effects. For this, cross-layer techniques combine accurate evaluation of the soft error resilience with a broad cross-layer exploration of different combinations of protection techniques. This work demonstrates how to apply the cross-layer resilience principle on custom processors, fixed-hardware processors, accelerators, and SRAM memories with a focus on soft errors. Its main focus spans from application to circuit layer as illustrated in Fig 1. These works lead

to a range of key insights, important for realizing cross-layer soft error resilience for a wide range of system components:

- accurate resilience evaluation is key, e.g., simulation-based fault injection at the flip-flop level is required to accurately evaluate soft errors in logic,
- multi-level/mixed-mode simulation enables very efficient resilience evaluation using fault injection,
- cross-layer resilience exploration must be customized for the component under consideration such as a custom processor, uncore components, third-party processor, accelerator, or SRAM,
- embedded applications such as control algorithms have inherent fault resilience that can be exploited,
- circuit-level techniques are crucial for cost-effective error resilience solutions, and
- existing architecture- and software-level techniques for hardware error resilience are generally expensive or provide too little resilience when implemented using their low-cost variants.

The chapter is structured as follows: first, evaluation methods using fault injection are covered, followed by cross-layer resilience exploration. Finally, experimental results are provided.

2 Evaluation of Soft Error Resilience Using Fault Injection

Fault injection is commonly used to evaluate soft error resilience. Radiation-induced soft errors can be modeled as bit flips [23], which are injected into the system's memory cells such as flip-flops and SRAM cells. There exists a wide range of fault injection methods, which will briefly be discussed in the following.

2.1 Overview on Fault Injection Methods

Hardware-based fault injection injects the fault in a hardware prototype of the system. For example, a radiation beam experiment can be used to provoke faults in an ASIC. This is a very expensive experimental setup, e.g., requiring a radiation source such as used in [1]. The chip hardware can also be synthesized to an FPGA, which is instrumented with additional logic to change bit values in the memory, flip-flops, or combinational paths of the logic to inject a fault using *emulation-based fault injection* [10, 13]. Embedded processors have a debug port to read out their internal states such as architectural registers. These debug ports often also enable the ability to change the internal states. This can be used to inject a fault in the processor using *debug-based fault injection* [15, 41]. Software running on the system can be used to mimic faults in *software-implemented fault injection*, e.g., as

presented in [26, 30, 44]. The compiler can be used to instrument the binary with fault injection code, for *compiler-based fault injection*, e.g., implemented in [18]. *Simulation-based fault injection* injects faults in a simulation model of the system. It is commonly applied to investigate the error resilience of the system and, hence, is the primary focus of this work.

2.2 Simulation-Based Fault Injection

Simulation-based fault injection provides very good properties in terms of parallelism, observability, and early availability during the design. Simulation-based fault injection can be realized at different levels of abstraction. For *gate-level fault injection*, the fault is injected into the gate Netlist of the system obtained after logic synthesis. For *flip-flop-level fault injection*, the fault is injected into the RTL implementation of the system. The fault impact is simulated using logic simulation, e.g., as used in [12, 46]. In *architectural-level fault injection*, the fault is injected either in a micro-architectural simulator or Instruction Set Simulator (ISS). Micro-architectural simulators such as Gem5 [3] simulate all architectural and some micro-architectural states such as pipeline registers of the processor, e.g., as presented in [25], but usually do not accurately model the processor's control logic. An ISS usually only simulates the architectural registers, but not any micro-architectural registers. ISSs are used for fault injection in [14, 24, 35]. In *software-level fault injection*, the fault is directly injected into a variable of the executing program. The software can then be executed to determine the impact of the corrupted variable on the program outputs.

A key insight of previous work was that the evaluation of the soft error resilience of logic circuits such as processor pipelines requires flip-flop-level fault injection, e.g., using the RTL model [9, 38]. Architectural-level and software-level fault injection may not yield accurate results as they do not include all details of the logic implementation as will also be shown in the results in Sect. 4.1. In contrast, soft errors in memories such as SRAM may be investigated at architectural level, which models memory arrays in a bit-accurate fashion.

2.3 Fast Fault Injection for Processor Cores

A good estimation of soft error resilience requires simulating a large amount of fault injection scenarios. This may become computationally infeasible when long-running workloads are evaluated, e.g., for embedded applications. Such long test cases arise in many applications. For example, in order to evaluate the impact of a soft error on a robotic control application, the control behavior needs to simulate several seconds real time, possibly simulating several billion cycles of the digital hardware. An efficient analysis method called ETISS-ML for evaluating the

resilience against soft errors in the logic of a processor sub-system is presented in [37, 38]. A typical processor sub-system of a micro-controller consists of the pipeline, control path, exception unit, timer, and interrupt controller. ETISS-ML is especially efficient for evaluating the impact of soft errors for long software test cases.

2.3.1 Multi-Level Fault Injection

ETISS-ML reduces the computational cost of each fault injection run by applying a multi-level simulation approach, which was also applied in other fault injection environments such as [16, 31, 45]. The key idea is to switch abstraction of the processor model during the fault injection run and to minimize the number of cycles simulated at flip-flop level. For this, an ISS is used in addition to the RTL model of the processor at flip-flop level.

The proposed multi-level flow is illustrated in Fig. 2. First the system is booted in ISS mode. This allows to quickly simulate close to the point of the fault injection, at which point, the simulation switches to flip-flop-level. During the RTL warmup phase, instructions are executed to fill the unknown micro-architectural states of the processor sub-system. This is required as the architectural registers are not visible to the ISS simulation. After this RTL warmup, the fault is injected as a bit flip. During the following RTL cool-down phase, the propagation of the fault is tracked. Once the initial impact of the fault propagates out of the processor's micro-architecture or is masked, the simulation can switch back to ISS mode. ETISS-ML reaches between 40x-100x speedup for embedded applications compared to pure flip-flop-level fault injection while providing the same accuracy [37, 38].

Both the switch from ISS mode to RTL mode as well as the switch from RTL to ISS mode require careful consideration. If a simulation artifact (wrong behavior) is produced by the switching process, it may be wrongly classified as fault impact.

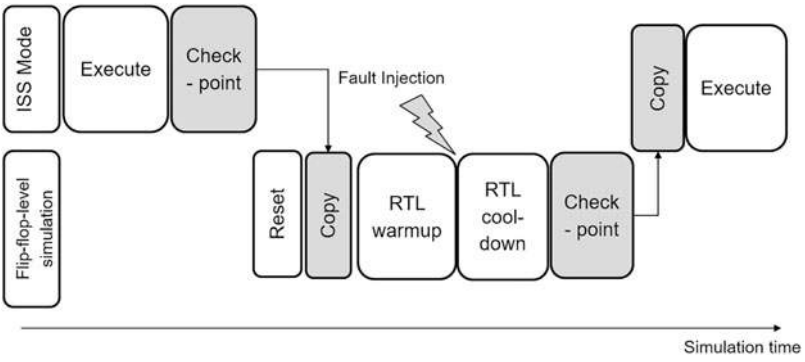


Fig. 2 Multi-level simulation flow of ETISS-ML

Next, we will detail the state of the art approach used by ETISS-ML to solve these challenges.

2.3.2 Switch from ISS Mode to Flip-Flop-Level Simulation

As shown in Fig. 2, a checkpoint is taken from the ISS to initialize the state in the RTL processor model. This checkpoint only includes the architectural states, the micro-architectural states such as pipeline registers are unknown. In the RTL warmup phase instructions are executed to fill up these micro-architectural states. In order to verify the RTL warmup phase, a (0, 1, X) logic simulation can be applied [37]. All micro-architectural states are initialized to X (unknown), while the values of architectural states are copied from the checkpoint. Additionally, the inputs loaded from external devices such as instruction and data memories as well as peripheral devices are also known from ISS simulation. Naturally, one expects that the micro-architectural states take known values after a certain number of instructions are executed. A key insight here was that this is not the case. Several state machines in the control path and bus interfaces of the processor would start from an unknown state. Hence, all following states remain unknown. One must assume initial states for the RTL state machines, e.g., the reset state. Then one can observe the removal of X values in the RTL model to derive a suitable RTL warmup length for a given processor architecture.

2.3.3 Switch from Flip-Flop-Level Simulation Back to ISS Mode

After the fault has been injected into the RTL model, the flip-flop level simulation is continued during the RTL cool-down phase. When switching back to ISS mode, all micro-architectural states are lost, as only the architectural states are copied over. Hence, one must ensure that one does not lose information about the impact of the fault as this would result in an incorrect estimation. One can take a fixed, very long cool-down phase as proposed in [45]. Yet, this leads to inefficient simulation as many cycles need to be evaluated at flip-flop level. Additionally, one does not gain information as to whether or not the soft error impact is still present in the micro-architectural states. This can be improved by simulating two copies of the RTL model, a faulty processor model and a tracking model [38]. The external state of memories, peripherals, or the environment is not duplicated. The soft error is only injected into the faulty model. In contrast, the tracking model simulates without the error. Writes to the external devices (memories, peripherals) are only committed from the faulty model. Reads from those devices are supplied to both models. Hence, when the soft error is not masked, it may propagate from the faulty model to the architectural state, external memories and devices and, then, be read back to the faulty and tracking model. Whenever both models have the same micro-architectural state, one can be sure that the error either has been masked or has propagated fully to the architectural state or external devices and memories. At this point the simulation

can switch to ISS mode as the architectural state and external devices and memories are also modeled at ISS level. It turns out that some errors never propagate out of the micro-architectural states, e.g., because a configuration is corrupted that is never rewritten by the software. In this case the switch back to ISS mode is not possible as it would cause inaccuracies, e.g., as would be observed with a fixed cool-down length.

2.4 Fast Fault Injection in Uncore Components

In addition to errors impacting processor cores, it is equally important to consider the impact of errors in uncore components, such as cache, memory, and I/O controllers, as well. In SoCs, uncore components are comparable to processor cores in terms of overall chip area and power [33], and can have significant impact on the overall system reliability [8].

Mixed-mode simulation platforms are effective for studying the system-level impact and behavior of soft errors in uncore components as well. As presented in [8], such a platform would achieve a 20,000× speedup over RTL-only injection while ensuring accurate modeling of soft errors. Full-length applications benchmarks can be analyzed by simulating processor cores and uncore components using an instruction-set simulator in an accelerated mode. At the time of injection, the simulation platform would then enter a co-simulation mode, where the target uncore component is simulated using accurate RTL simulation. Once co-simulation is no longer needed (i.e., all states can be mapped back to high-level models), the accelerated mode can resume, allowing application benchmarks to be run to completion.

2.5 Fast Fault Injection for SRAM Memories Using Mixture Importance Sampling

Memories such as on-chip SRAM or caches are already modeled bit-accurately at micro-architectural and instruction-level. Hence, for the evaluation of soft errors in memories, fault injection into faster instruction-level models is possible. Yet, modern SRAMs are very dense such that the probability of multi-bit upsets (MBUs) due to soft errors is not negligible. For MBU fault models, straightforward Monte Carlo simulation requires a large sample size in the range of millions of sample elements to obtain sufficient confidence bounds.

To address this challenge one can apply mixture importance sampling to connect a technology-level fault model with a system-level fault simulation [29]. This propagation of low-level information to the system level is motivated by the Resilience Articulation Point (RAP) approach proposed in [23]. The key idea behind

RAP is that errors in the system should be modeled by probabilistic functions describing MBU's bit flip probabilities including spatial and temporal correlations. Thus, the impact of errors in the system can be evaluated, while maintaining a direct connection to their root causes at the technology level. The sample size to estimate the resilience of the system to soft errors in SRAMs can be massively reduced by guiding the Monte Carlo simulation to important areas. As an illustrative example, we assume that the SRAM is used to realize a data cache with 1-bit parity protection. MBUs that alter an odd number of bits in a cache line are detected by the parity checks and may be corrected by loading the correct value from the next level of memory. MBUs that alter an even number of bits in a cache line remain undetected and may cause silent data corruption. Additionally, MBUs may perturb several neighboring cache lines due to different MBU mechanisms. This can lead to mixed cases of recoverable errors and silent data corruption. For a cache with one bit parity protection, MBUs with even number $(2, 4, \ldots)$ of bits in one cache line are critical as they may provoke silent data corruption (SDC). The sampling strategy can be biased towards these MBUs by mixture important sampling, which speeds up the resilience evaluation significantly. It is shown that results with high confidence can be obtained with sample sizes in the thousands instead of millions [29]. The resulting fast evaluation enables the efficient exploration of the most efficient cross-layer protection mechanisms for the SRAM memory for an overall optimized reliable system.

3 Cross-Layer Exploration of Soft Error Resilience Techniques

Most safety-critical systems already employ protection techniques against soft errors at different layers. Yet often, possible combinations are not systematically explored and evaluated to identify a low-cost solution. This may result in inefficient redundancy and hardening, e.g., that certain types of faults are detected by multiple techniques at different layers, or certain redundancy is not required, as the circuit is adequately protected (e.g., by circuit-hardening techniques).

In this section several approaches are outlined that focus on cross-layer exploration for finding low-cost soft error protection:

- the CLEAR approach can generate resilience solutions for custom processors with selective hardening in combination with architectural and software-level protection schemes.
- Using a similar approach, on-chip SRAM can be protected with a combination of hardening and error detection codes.
- For third-party processors, hardening and hardware redundancy are not an option. Hence, we show how application resilience can be used in combination with software-level protection to achieve cross-layer resilience.

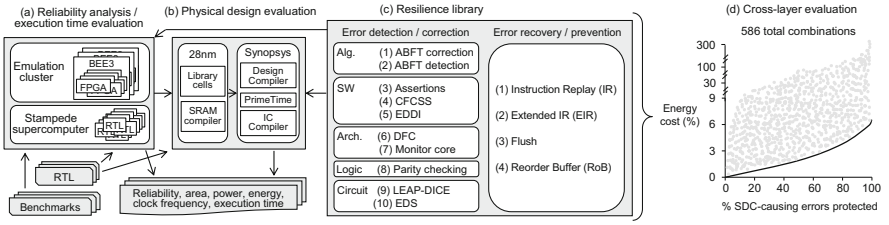


Fig. 3 CLEAR framework: (**a**) BEE3 emulation cluster/Stampede supercomputer injects over 9 million errors into two diverse processor architectures running 18 full-length application benchmarks. (**b**) Accurate physical design evaluation accounts for resilience overheads. (**c**) Comprehensive resilience library consisting of ten error detection/correction techniques + four hardware error recovery techniques. (**d**) Example illustrating thorough exploration of 586 cross-layer combinations with varying energy costs vs. percentage of SDC-causing errors protected

- Finally, we also discuss how accelerators can be protected with cross-layer resilience techniques.

3.1 CLEAR: Cross-Layer Resilience for Custom Processors

CLEAR (Cross-Layer Exploration for Architecting Resilience) is a first of its kind framework to address the challenge of designing robust digital systems: given a set of resilience techniques at various abstraction layers (circuit, logic, architecture, software, algorithm), how does one protect a given design from radiation-induced soft errors using (perhaps) a combination of these techniques, across multiple abstraction layers, such that overall soft error resilience targets are met at minimal costs (energy, power, execution time, area)?

CLEAR has broad applicability and is effective across a wide range of diverse hardware designs ranging from in-order (InO-core) and out-of-order (OoO-core) processor cores to uncore components such as cache controllers and memory controllers to domain-specific hardware accelerators. CLEAR provides the ability to perform extensive explorations of cross-layer combinations across a rich library of resilience techniques and error sources.

Figure 3 gives an overview of the CLEAR framework. Individual components are described briefly in the following:

3.1.1 Reliability Analysis

While the CLEAR framework provides the ability to analyze the reliability of designs, this component does not comprise the entirety of the framework. The modularity of the CLEAR framework enables one to make use of any number of the accurate fault-injection simulation components described in detail in Sect. 2.2

to perform reliability analysis. The analysis considered in this chapter encompasses both Silent Data Corruption (SDC) and Detected but Uncorrected Errors (DUE).

3.1.2 Execution Time Evaluation

Execution time is measured using FPGA emulation and RTL simulation. Applications are run to completion to accurately capture the execution time of an unprotected design. For resilience techniques at the circuit and logic levels, CLEAR ensures that modifications incorporating such resilience techniques will maintain the same clock speed as the unprotected design. For resilience techniques at the architecture, software, and algorithm levels, the error-free execution time impact is also reported.

3.1.3 Physical Design Evaluation

To accurately capture overheads associated with implementing resilience techniques, it is crucial to have a means for running an entire physical design flow to properly evaluate the resulting designs. To that end, the Synopsys design tools (Design Compiler, IC compiler, PrimeTime, and PrimePower) with a commercial 28nm technology library (with corresponding SRAM compiler) are used to perform synthesis, place-and-route, and power analysis. Synthesis and place-and-route (SP&R) is run for all configurations of the design (before and after adding resilience techniques) to ensure all constraints of the original design (e.g., timing and physical design) are met for the resilient designs as well.

3.1.4 Resilience Library

For processor cores, ten error detection and correction techniques together with four hardware error recovery techniques are carefully chosen for analysis. In the context of soft error resilience, error detection and correction techniques include: Algorithm Based Fault Tolerance (ABFT) correction, ABFT detection, Software assertions, Control Flow Checking by Software Signatures (CFCSS), Error Detection by Duplicated Instructions (EDDI), Data Flow Checking (DFC), Monitor cores, Parity checking, flip-flop hardening using LEAP-DICE, and Error Detection Sequential (EDS). These techniques largely cover the space of existing soft error resilience techniques. The characteristics (e.g., costs, resilience improvement, etc.) of each technique when used as a standalone solution (e.g., an error detection/correction technique by itself or, optionally, in conjunction with a recovery technique) are presented in Table 1. Additionally, four micro-architectural recovery techniques are included: Instruction Replay (IR), Extended IR (EIR), flush, and Reorder Buffer (RoB) recovery. Refer to [7] for an in-depth discussion of specific techniques and their optimizations, including a detailed discussion of Table 1.

Table 1 Individual resilience techniques: costs and improvements as a standalone solution

Layer	Technique		Area cost	Power cost	Energy cost	Exec. time impact	Avg. SDC improve	Avg. DUE improve	False positive	Detection latency	γ
Circuit	LEAP-DICE (no additional recovery needed)	InO	0–9.3%	0–22.4%	0–22.4%	0%	1 × −5000×	1 × −5000×	0%	N/A	1
		OoO	0–6.5%	0–9.4%	0–9.4%						
	EDS (without recovery—unconstrained)	InO	0–10.7%	0–22.9%	0–22.9%	0%	1 × −>100,000×	0.1 × −1×	0%	1 cycle	1
		OoO	0–12.2%	0–11.5%	0–11.5%						
	EDS (with IR recovery)	InO	0–16.7%	0–43.9%	0–43.9%	0%	1 × −>100,000×	1×−>100,000×	0%	1 cycle	1.4
		OoO	0–12.3%	0–11.6%	0–11.6%						1.06
Logic	Parity (without recovery—unconstrained)	InO	0–10.9%	0–23.1%	0–23.1%	0%	1 × −>100,000×	0.1 × −1×	0%	1 cycle	1
		OoO	0–14.1%	0–13.6%	0–13.6%						
	Parity (with IR recovery)	InO	0–26.9%	0–44%	0–44%	0%	1 × −>100,000×	1×−>100,000×	0%	1 cycle	1.4
		OoO	0–14.2%	0–13.7%	0–13.7%						1.06
Arch.	DFC (without recovery—unconstrained)	InO	3%	1%	7.3%	6.2%	1.2×	0.5×	0%	15 cycles	1.28
		OoO	0.2%	0.1%	7.2%	7.1%					1.09
	DFC (with EIR recovery)	InO	37%	33%	41.2%	6.2%	1.2×	1.4×	0%	15 cycles	1.48
		OoO	0.4%	0.2%	7.3%	7.1%					1.14
	Monitor core (with RoB recovery)	OoO	9%	16.3%	16.3%	0%	19×	15×	0%	128 cycles	1.38

(continued)

Table 1 (continued)

Layer	Technique		Area cost	Power cost	Energy cost	Exec. time impact	Avg. SDC improve	Avg. DUE improve	False positive	Detection latency	γ
Software	Software assertions for general-purpose processors (without recovery—unconstrained)	InO	0%	0%	15.6%	15.6%	1.5×	0.6×	0.003%	9.3M cycles	1.16
	CFCSS (without recovery—unconstrained)	InO	0%	0%	40.6%	40.6%	1.5×	0.5×	0%	6.2M cycles	1.41
	EDDI (without recovery—unconstrained)	InO	0%	0%	110%	110%	37.8×	0.3×	0%	287K cycles	2.1
Alg.	ABFT correction (no additional recovery needed)	InO	0%	0%	1.4%	1.4%	4.3×	1.2×	0%	N/A	1.01
		OoO									
	ABFT detection (without recovery—unconstrained)	InO	0%	0%	24%	1–56.9%	3.5×	0.5×	0%	9.6M cycles	1.24
		OoO									

3.1.5 Exploration

CLEAR approaches cross-layer exploration using a top-down approach: resilience techniques from upper layers of the resilience stack (e.g., algorithm-level techniques) are applied before incrementally moving to lower layers (e.g., circuit-level techniques). This approach helps generate cost-effective solutions that leverage effective interactions between techniques across layers. In particular, while resilience techniques from the algorithm, software, and architecture layers of the stack generally protect multiple flip-flops, a designer typically has little control over the specific subset of flip-flops that will be protected. Using multiple techniques from these layers can lead to a situation where a given flip-flop may be protected (sometimes unnecessarily) by multiple techniques. Conversely, resilience techniques at the logic and circuit layers offer fine-grained protection since these techniques can be applied selectively to individual flip-flops (i.e., flip-flops not (sufficiently) protected by higher-level techniques).

3.2 Resilience Exploration for Custom Accelerators

Domain-specific hardware accelerators will increasingly be integrated into digital systems due to their ability to provide more energy-efficient computation for specific kernels. As a result of their application-specific nature, hardware accelerators have the opportunity to leverage application space constraints when exploring cross-layer resilience (i.e., resilience improvement targets only need to hold over a limited subset of applications). Accelerators also benefit from the ability to create natural checkpoints for recovery by protecting the memory storing the accelerator inputs (e.g., using ECC), allowing for a simple means for re-execution on error detection. Therefore, the cross-layer solutions that provide cost-effective resilience may differ from those of processor cores and warrant further exploration.

3.3 Cross-Layer Resilience for Exploration for SRAM Memories

In [28], a cross-layer approach for soft error resilience was applied to SRAM data caches. Again, a systematic exploration requires having a good evaluation of the cost and efficiency of the applied protection mechanisms. In this study, the available protection mechanisms were the following: at circuit level, either (1) the supply voltage could be raised by 10% or (2) the SRAM cells could be hardened by doubling the area. At the architectural level, (3) 1-bit parity could be introduced in the cache lines. The circuit-level hardening techniques require parameterizing the statistical MBU fault model introduced in Sect. 2.5 considering cell area, supply

voltage and temperature. For each configuration, the fault probabilities for MBU patterns need to be evaluated to obtain a good estimate of soft error probabilities. Additionally, the architecture and workload play a key role in the evaluation as not all soft errors are read from the cache. Here again, architectural-level simulation can be used to simulate the workload using fault injection into a bit-accurate cache model.

3.4 Towards Cross-Layer Resiliency for Cyber-Physical Systems (CPS)

In benchmark-type workloads, silent data corruption in a single program output commonly leads to a failure, e.g., an encryption algorithm fails if its encrypted data is corrupted such that it cannot be decrypted. Hence, cross-layer resiliency often targets reducing the rate of silent data corruption.

For cyber-physical systems (CPS), however, many workloads can tolerate deviations from the fault-free outcome, e.g., in an embedded control algorithm, noise, e.g., in sensors, is present and considered in the control design. It will treat silent data corruption as yet another noise source, that can, possibly, be tolerated for minor deviations from the correct value. Another effect is that CPS workloads are commonly scheduled as periodic tasks. Often, the outputs of one instance of a certain task are overwritten by the next instance of a task. Hence, a corruption of the output of a single task has an effect only for a certain duration in time. Subsequent task executions might mitigate the effect of silent data corruption before the system behavior becomes critical. For example for control applications, the sampling rate of the controller is often higher than demanded, such that a single corrupted actuation command will not lead to a failure within one control period. Following sensor readouts will show a deviation from the desired control behavior that is corrected by the controller in subsequent control periods.

In order to consider the inherent resilience of CPS workloads, a full system simulation is required. CPS usually form a closed loop with their environment, e.g., actuation will change the physical system behavior, which determines future sensor readouts. Extensive fault injection for obtaining a good resiliency evaluation is enabled by the fast simulation speed of ETISS-ML [38], while RTL level fault injection would be prohibitively slow to evaluate system behavior over a long system-level simulation scenario. ETISS-ML can be integrated into a full-system virtual prototype (VP) that models the system and its physical environment such that error impacts can be classified considering the inherent resilience of CPS workloads. For this, the physical behavior is traced to determine the impact of the error. A major question to be investigated is how this inherent application resilience can be exploited in an efficient way to reduce cost of protection techniques towards cross-layer resilience of CPS.

4 Experimental Results

This section presents results for cross-layer exploration. First, we show results that support our claim that flip-flop level fault injection is required for soft errors in logic. Then we provide the results for cross-layer exploration with CLEAR and ETISS-ML for processors. Finally, we show the results for the cross-layer exploration of protection techniques for the data cache of a control system for a self-balancing robot.

4.1 Accuracy of FI at Different Abstraction Levels

For radiation-induced soft errors, flip-flop soft error injection is considered to be highly accurate. Radiation test results confirm that injection of single bit flips into flip-flops closely models soft error behaviors in actual systems [4, 43]. On the other hand, [9] has shown that naïve high-level error injections (e.g., injection of a single-bit error into an architecture register, software-visible register-file, or program variable) can be highly inaccurate.

Accurate fault-injection is crucial for cost-effective application of cross-layer resilience. Inaccurate reliability characterization may lead to over- or underprotection of the system. Overprotection results in wasted cost (e.g., area, power, energy, price) and underprotection may result in unmitigated system failures.

In order to observe the impact of soft errors in the data and control path of a OR1K processor sub-system, the error propagation was tracked to the architectural-visible states in [38] for four test cases. In total 70k fault injection scenarios were run on each test case. The injection points were micro-architectural FFs in the RTL implementation such as pipeline and control path registers, that are not visible at the architectural level. First all soft errors were identified that had no impact on the architectural state since they were either being masked or latent. On average these were 67.51%.

On architectural level, we inject single bit flip fault scenarios as it is unclear what multi-bit fault scenarios could really happen in HW. These scenarios will cover all single bit flip soft errors in an architectural state as well as any soft error in a micro-architectural state that propagates and corrupts just a single bit of an architectural state. In this case it makes no difference whether we inject the single bit flip in the micro-architectural state or architectural state. Yet, the distribution could be different. We now observe the experimental results as given in Table 2: 25.09% of the micro-architectural faults corrupted a single bit in the architectural state for a single cycle. These faults would be covered by fault injection at architectural level. But 7.40% of the soft errors corrupted several bits of the architectural state or lead to several bit flips in subsequent cycles. Injecting single bit soft errors in architectural states at architecture- or software level will not cover these micro-architectural fault

Table 2 Impact of single bit flip in micro-arch FFs on architectural processor state

Test case	Masked or latent [%]	Single bit corruption [%]	Multi-bit corruption [%]
JDCT	66.77	25.68	7.55
AES	66.36	26.13	7.51
IIR	68.88	23.83	7.29
EDGE	68.02	24.73	7.25
Average	67.51	25.09	7.40

Table 3 General-purpose processor core designs studied

	Design	Description	Clk. freq.	Error injections	Instructions per cycle
InO	LEON3 [17]	Simple, in-order (1250 flip-flops)	2.0 GHz	5.9 million	0.4
OoO	IVM [46]	Complex, super-scalar, out-of-order (13,819 flip-flops)	600 MHz	3.5 million	1.3

scenarios. Hence, one needs to look into RTL fault injection to obtain accurate results for these faults.

4.2 Cross-Layer Resilience Exploration with CLEAR

The CLEAR framework is first used to explore a total of 586 cross-layer combinations in the context of general-purpose processor cores. In particular, this extensive exploration consists of over 9 million flip-flop soft error injections into two diverse processor core architectures (Table 3): a simple, in-order SPARC LEON3 core (InO-core) and a complex superscalar out-of-order Alpha IVM core (OoO-core). Evaluation is performed across 18 application benchmarks from the SPECINT2000 [22] and DARPA PERFECT [2] suites.

Several insights resulted from this extensive exploration: accurate flip-flop level injection and layout (i.e., physical design) evaluation reveal many individual techniques provide minimal (less than 1.5×) SDC/DUE improvement (contrary to conclusions reported in the literature that were derived using inaccurate architecture- or software-level injection [20, 36]), have high costs, or both. The consequence of this revelation is that most cross-layer combinations have high cost.

Among the 586 cross-layer combinations explored using CLEAR, a highly promising approach combines selective circuit-level hardening using LEAP-DICE, logic parity, and micro-architectural recovery (flush recovery for InO-cores, reorder

buffer (RoB) recovery for OoO-cores). Thorough error injection using application benchmarks plays a critical role in selecting the flip-flops protected using these techniques.

From Table 4, to achieve a 50× SDC improvement, the combination of LEAP-DICE, logic parity, and micro-architectural recovery provides 1.5× and 1.2× energy savings for the OoO- and InO-cores, respectively, compared to selective circuit hardening using LEAP-DICE. This scenario is shown under "bounded latency recovery." The relative benefits are consistent across benchmarks and over the range of SDC/DUE improvements.

If recovery hardware is not needed (i.e., there exist no recovery latency constraints and errors can be recovered using an external means once detected), minimal (<0.2% energy) savings can be achieved when targeting SDC improvement. This scenario is shown under "unconstrained recovery." However, without recovery hardware, DUEs increase since detected errors are now uncorrectable; thus, no DUE improvement is achievable.

Additional cross-layer combinations spanning circuit, logic, architecture, and software layers are presented in Table 4. In general, most cross-layer combinations are not cost-effective. For general-purpose processors, a cross-layer combination of LEAP-DICE, logic parity, and micro-architectural recovery provides the lowest cost solution for InO- and OoO-cores for all improvements.

4.3 Resilience Exploration for Custom Accelerators

Utilizing a high-level synthesis (HLS) engine from UIUC [5], 12 accelerator designs derived from the PolyBench benchmark suite [42] were evaluated with protection using LEAP-DICE (circuit), logic parity (logic), modulo-3 shadow datapaths (architecture), EDDI (software), and ABFT (algorithm) techniques. Note that, software and algorithm techniques are converted into hardware checkers during high-level synthesis.

Consistent with processor core results, cost-effective resilience solutions for domain-specific hardware accelerators (Table 5) required the use of circuit-level techniques (e.g., a 50× SDC improvement was achieved at less than 6% energy cost using a combination of application-guided selective LEAP-DICE and logic parity). However, even given the application-constrained context of accelerators, software-level (and algorithm-level) resilience techniques were unable to provide additional benefits.

Table 4 Costs vs. SDC (DUE) improvements for various combinations in general-purpose processors

			Bounded latency recovery										Unconstrained recovery										Exec. time impact
			SDC improvement					DUE improvement					SDC improvement					DUE improvement					
			2	5	50	500	Max	2	5	50	500	Max	2	5	50	500	Max	2	5	50	500	Max	
InO	LEAP-DICE + logic parity (+ flush recovery)	A	0.7	1.7	2.5	3	8	0.6	1.5	3.6	4.4	8	0.7	1.6	2.4	2.8	7.6	–	–	–	–	–	0%
		P	1.9	3.9	6.1	6.7	17.9	1.5	3.4	8.4	10.4	17.9	1.9	3.8	5.9	6.5	17.2						
		E	1.9	3.9	6.1	6.7	17.9	1.5	3.4	8.4	10.4	17.9	1.9	3.8	5.9	6.5	17.2						
	EDS + LEAP-DICE + logic parity (+ flush recovery)	A	0.9	2.3	2.7	3.3	8.4	0.8	2.1	3.8	4.8	8.4	0.9	2.2	2.5	3.2	8.1	–	–	–	–	–	0%
		P	1.9	4.3	6.6	7.2	19.3	1.7	3.8	8.5	11	19.3	1.9	4.2	6.3	7.1	19						
		E	1.9	4.3	6.6	7.2	19.3	1.7	3.8	8.5	11	19.3	1.9	4.2	6.3	7.1	19						
	DFC + LEAP-DICE + logic parity (+ EIR recovery)	A	39.3	41.1	41.5	43.1	45	39.3	39.9	41.9	42.5	45	3.3	5.1	5.6	7.1	10.6	–	–	–	–	–	6.2%
		P	32.4	35.5	38.7	41	50.9	32.5	33.9	38.4	40.7	50.9	1.4	4.8	8.1	10	18.2						
		E	44.2	56.7	60.2	62.7	60.3	45.8	48.9	58.3	63	60.3	10.6	13.9	17.4	19.9	25.5						
	Assertions + LEAP-DICE + logic parity (no recovery)	A	–	–	–	–	–	–	–	–	–	–	0.7	0.9	1	1.1	7.6	–	–	–	–	–	15.6%
		P											1.4	1.8	2.2	2.2	17.2						
		E											17.1	17.5	18	18	24.5						
	CFCSS + LEAP-DICE + logic parity (no recovery)	A	–	–	–	–	–	–	–	–	–	–	0.3	1	1.4	1.3	7.6	–	–	–	–	–	40.6%
		P											0.8	1.8	2.9	3.1	17.2						
		E											41.5	43	44.6	44.9	146						
	CFCSS + LEAP-DICE + logic parity (no recovery)	A	–	–	–	–	–	–	–	–	–	–	0.3	1	1.4	1.3	7.6	–	–	–	–	–	40.6%
		P											0.8	1.8	2.9	3.1	17.2						
		E											41.5	43	44.6	44.9	146						
	EDDI + LEAP-DICE + logic parity (no recovery)	A	–	–	–	–	–	–	–	–	–	–	0	0	0.7	0.9	7.6	–	–	–	–	–	110%
		P											0	0	0.6	0.8	17.2						
		E											110	110	111	111	146						

OoO	LEAP-DICE + logic parity (+ RoB recovery)	A	0.06	0.1	1.4	2.2	4.9	0.5	0.7	2.6	3	4.9	0.06	0.1	1.4	2.2	4.9	–	–	–	–	–	0%
		P	0.1	0.2	2.1	2.4	7	0.1	0.1	2	1.8	7	0.1	0.2	2.1	2.4	7						
		E	0.1	0.2	2.1	2.4	7	0.1	0.1	2	1.8	7	0.1	0.2	2.1	2.4	7						
	EDS + LEAP-DICE + logic parity (+ RoB recovery)	A	0.07	0.1	1.6	2.2	5.4	0.6	0.8	2.6	3	5.4	0.07	0.1	1.6	2.2	5.4	–	–	–	–	–	0%
		P	0.1	0.2	2.3	2.5	8.1	0.1	0.1	2	1.8	8.1	0.1	0.2	2.3	2.5	8.1						
		E	0.1	0.2	2.3	2.5	8.1	0.1	0.1	2	1.8	8.1	0.1	0.2	2.3	2.5	8.1						
	DFC + LEAP-DICE + logic parity (+ EIR recovery)	A	0.2	1	1.8	2	5.3	0.2	0.4	1.7	3.9	5.3	0.1	0.8	1.6	1.8	5.1	–	–	–	–	–	7.1%
		P	1.1	1.4	2	2.8	7.2	0.2	0.2	2.6	3.3	7.2	1	1.3	1.9	2.7	7.1						
		E	21.2	21.5	22.2	23	14.8	20	20.1	22.9	23.6	14.8	10	11.4	12.1	12.9	14.7						
	Monitor core + LEAP-DICE + logic parity (+ RoB rec.)	A	9	9	9.8	10.5	13.9	9	9	10.1	11.2	13.9	9	9	9.8	10.5	13.9	–	–	–	–	–	0%
		P	16.3	16.3	20	20.2	23.3	16.3	16.3	20.1	21.5	22.3	16.3	16.3	20	20.2	23.3						
		E	16.3	16.3	20	20.2	23.3	16.3	16.3	20.1	21.5	22.3	16.3	16.3	20	20.2	23.3						

A (area cost %), P (power cost %), E (energy cost %)

Table 5 Costs (area/energy) and improvements for resilience in 12 domain-specific accelerators

	SDC improvement			
Resilience technique(s)	2×	5×	50×	500×
Selective LEAP-DICE	0.9%/3.3%	1.2%/5%	1.7%/7%	2.2%/8.8%
Selective parity checking	1.4%/4.4%	2.2%/6.4%	3.1%/8.7%	3.4%/10.6%
LEAP-DICE + parity	0.6%/2.7%	1%/3.9%	1.3%/5.7%	1.7%/7.4%
Mod-3 + LEAP-DICE + parity	0.7%/3.6%	2.3%/4.7%	2.9%/6.5%	3.3%/8.1%
EDDI + LEAP-DICE + parity	27.6%/33%	27.6%/33.2%	27.6%/33.4%	28.3%/34%
ABFT + LEAP-DICE + parity	11.9%/23.8%	12.2%/24.1%	12.3%/24.2%	12.3%/24.8%

Table 6 Micro-controller (μC) design studied

	Design	Description	Clk. freq.	Error injections
μC	OpenRISC [40]	Simple, in-order (no caches), (1440 flip-flops) with timer and interrupt controller	100 MHz	500,000

4.4 Resilience Exploration for Fixed-hardware Micro-Controller

The multi-level simulation was implemented for a fixed-hardware micro-controller (μC) as shown in Table 6. The RTL implementation uses only the pipeline, programmable interrupt controller, and timer but no caches in order to have a μC-type processor similar to ARM's Cortex M family. We study a full system simulation setup based on a SystemC VP, which models an μC used in a simplified adaptive cruise control (ACC) system. Its goal is to maintain a constant distance between two moving vehicles by controlling the speed of the rear vehicle via the throttle value of the motor (actuator). The processor of the μC periodically executes a PI control algorithm. The PI control algorithm's inputs are sensor values measuring the distance to the front vehicle and speed of the rear vehicle. Figure 4 shows the SystemC/TLM model structure of the system with μC, actuator and sensors. The sensor values are dynamically generated by a physics simulation of the two vehicles based on the commands sent to the actuator. The system boots and then starts execution from time zero. We define a simple safety specification to demonstrate the evaluation. The desired distance between the vehicles is set to 40 m. A fault is classified to cause a system-level failure when the distance leaves the corridor between 20 m and 60 m within a given driving scenario. For this scenario, both vehicles have same speed and a distance of 50 m at time zero.

Figure 5 shows the simulation results for four fault injection (FI) simulations. The green curve shows a soft error that has no influence on the system outputs, which results in the same curve visible in the fault-free run. The blue curve shows the inherent fault tolerance of control algorithms. Even though the actuator output is corrupted by the soft error, the control algorithm is able to recover from the

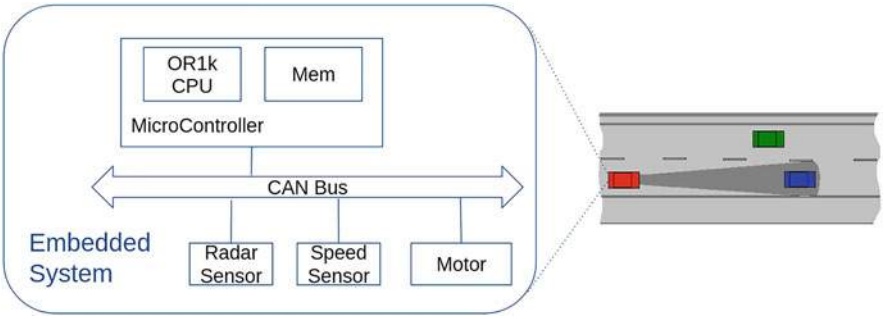


Fig. 4 SystemC VP of control system

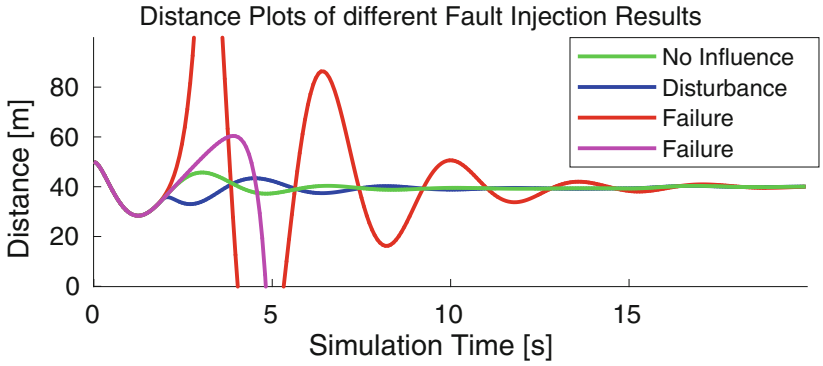


Fig. 5 Distance plotted for different FIs

disturbance. The distance does not leave the specified corridor. Finally, the pink and red curves show faults leading to a system failure.

In order to test cross-layer resiliency, we apply the following error detection and handling mechanisms. We concentrate on methods supported by fixed-hardware μCs, for which we would not be able to modify the logic or circuit implementation.

Watchdog Timer (WDT) The control algorithm has to write a value to the actuator every 10 ms. If no actuator write is detected, the system is reset by the WDT.

Task Duplication The control task is executed twice and the results are compared before the actuation.

EDDI EDDI is applied by the compiler to protect the data flow of the control application.

CFCSS CFCSS is applied by the compiler to protect the control flow of the control application.

The compiler can only apply EDDI and CFCSS on the software functions of the PI control task, not on software functions coming from the pre-compiled OR1K C-

Table 7 Comparison of resilience techniques for μC with watchdog timer (WDT) and external recovery by system reset

Resilience technique(s)	WDT Det. rate	SW Det. rate	SDC rate	Failure rate due to SDC	Exec. time impact
WDT	8.562%	0%	0.674%	0.061%	0%
Task duplic.+WDT	11.429%	1.284%	0.026%	0.002%	146.21%
EDDI+WDT	11.926%	1.706%	0.014%	0.002%	155.86%
CFCSS+WDT	8.929%	2.028%	0.542%	0.047%	0.249%
EDDI+CFCSS+WDT	13.370%	2.169%	0.017%	0.001%	156.857%

libraries. When task duplication, EDDI or CFCSS detect a fault, the SW triggers a reset.

Each method comes with a certain overhead and improvement in SDC rate as shown in Table 7. The column "WDT Det. Rate" shows the percentage of faults detected by the watchdog timer. The column "SW Det Rate" shows the percentage of faults detected by EDDI, CFCSS and the comparison for Task Duplication (depending on which protection is used). The SDC rate shows the percentage of faults that lead to a corrupt actuation value without being detected by a protection technique. Finally, the failure rate due to SDC shows the percentage of SDCs that lead to a failure of the control algorithm. Exec. Time Impact shows the overhead due to software redundancy inserted by the protection mechanisms. A WDT requires additional area, which is usually available on modern μCs, hence, this is ignored.

The following conclusions can be derived from the results: overall, the WDT detection rate is very high as it detects most DUEs, that result in incorrect timing of the application. EDDI and task duplication increases the execution time of the control task significantly at the cost of idle time of the processor. Yet, they also lead to significant SDC reduction. EDDI is slightly better, as it works on the intermediate representation (IR) and has a smaller vulnerability window. CFCSS also increases the software detection rate. Upon closer inspection, CFCSS does not lead to a significant reduction in SDC rate for both cases with and without EDDI. The application has a simple control flow, hence, control flow errors are rare. Most of the errors detected by CFCSS are due to errors during execution of the CFCSS check codes themselves. Hence, they would not lead to SDC of the functional code, yet, many errors are reported.

4.5 Resilience Exploration for SRAM Cache of Self-Balancing Robot

The cross-layer exploration was applied to a self-balancing robot system in [28] as shown in Fig. 6. The results are shown in Fig. 7. The figure shows the results for nominal SRAM design (N), increased supply voltage (V), increased area (A) and parity protection (P). The blue bar shows the rate of silent data corruption

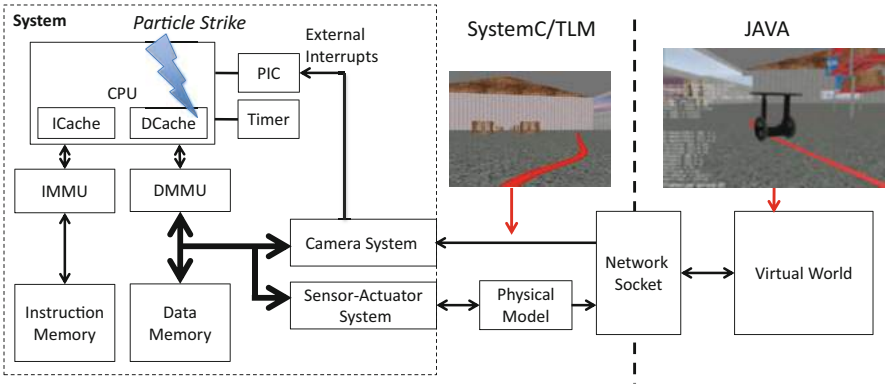


Fig. 6 Full simulation setup for self-balancing robot

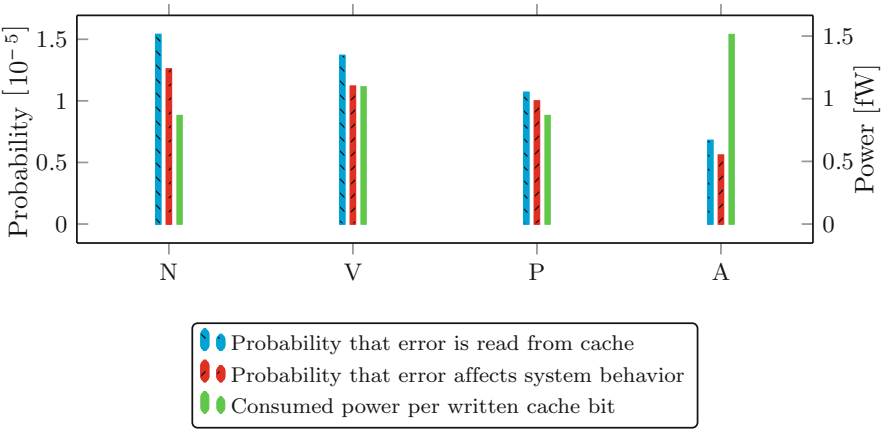


Fig. 7 Resilience exploration for cache of self-balancing robot

caused when a faulty cache line is read. The red bar shows those cases of silent data corruption that significantly affect the system behavior, which we classify as failure. The difference between the blue and red bar denotes the inherent resilience of the system. For hardening the system, increasing the supply voltage (V) decreases the silent data corruption rate (blue) and failure rate (red) but also increases the required power per written cache bit (green). Increasing the area (A) decreases the silent data corruption rate and failure rate more effectively compared to increasing the supply voltage but at the cost of a larger increase in power. In contrast, the parity protection (P) behaves differently to the hardening solutions. While parity also decreases the rate of silent data corruption (blue), we see that those remaining errors that are read from the cache (caused by an even number of upsets in the cache line) relatively often influence the system behavior (red), which is classified as failure. In the case of 1-bit parity protection the system is effectively protected from an odd number of

errors in each cache line. Yet, compared to the nominal case the failure probability of the system is only slightly reduced. The even number of upsets (mostly two bit upsets) are causing more often a failure than the detected single bit upsets. Upsets with three and more bits are not as relevant as they are very rare events. The key insight is that decreasing silent data corruptions thus does not necessarily result in a similar improvement in failure rate when considering the inherent resilience of the CPS application.

5 Conclusions

This chapter covered the fast evaluation of resilience against radiation-induced soft errors with multi-level/mixed-mode fault injection approaches as well as the systematic exploration of protection techniques that collaborate in a cross-layer fashion across the system stack. The methods were shown for case studies on custom processors, accelerators, third-party micro-controllers, and an SRAM-based cache.

Although this chapter has focused on radiation-induced soft errors, our cross-layer methodology and framework are equally effective at protecting against additional error sources such as supply voltage variations, early-life failures, circuit aging, and their combinations. For example, [6] demonstrates that cost-effective protection against supply voltage variation is achieved using Critical Path Monitor (CPM) circuit failure prediction and instruction throttling at 2.5% energy cost for a 64 in-order core design.

For error sources (such as early-life failures and circuit aging) that result from system degradation over longer duration of time (days to years), periodic on-line self-test and diagnostic are particularly effective at generating signatures to observe such degradation [27, 32, 34]. Since many of the resilience techniques considered in this chapter operate independently of the underlying error source, our conclusions regarding these particular techniques are broadly applicable.

Finally, an open question that remains is how to efficiently exploit the inherent resilience of CPS workloads. Full system simulation can help in a fast evaluation, but it remains to be seen in future research how the cost of resilience can be reduced by fully exploiting this potential in a cross-layer fashion.

References

1. Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., Leber, G.H.: Comparison of physical and software-implemented fault injection techniques. IEEE Trans. Comput. **52**(9), 1115–1133 (2003). https://doi.org/10.1109/TC.2003.1228509
2. Barker, K., Benson, T., Campbell, D., Ediger, D., Gioiosa, R., Hoisie, A., Kerbyson, D., Manzano, J., Marquez, A., Song, L., Tallent, N., Tumeo, A.: PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual. Pacific Northwest National Laboratory and Georgia Tech Research Institute (2013). http://hpc.pnnl.gov/projects/PERFECT/

3. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The GEM5 simulator. ACM SIGARCH Comput. Archit. News **39**(2), 1–7 (2011)
4. Bottoni, C., Glorieux, M., Daveau, J.M., Gasiot, G., Abouzeid, F., Clerc, S., Naviner, L., Roche, P.: Heavy ions test result on a 65 nm Sparc-V8 radiation-hard microprocessor. In: 2014 IEEE International Reliability Physics Symposium, pp. 5F.5.1–5F.5.6 (2014). https://doi.org/10.1109/IRPS.2014.6861096
5. Campbell, K.A., Vissa, P., Pan, D.Z., Chen, D.: High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2015). https://doi.org/10.1145/2744769.2744851
6. Cheng, E.: Cross-layer resilience to tolerate hardware errors in digital systems. Ph.D Dissertation, Stanford University (2018)
7. Cheng, E., Mirkhani, S., Szafaryn, L.G., Cher, C., Cho, H., Skadron, K., Stan, M.R., Lilja, K., Abraham, J.A., Bose, P., Mitra, S.: Tolerating soft errors in processor cores using clear (cross-layer exploration for architecting resilience). IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(9), 1839–1852 (2018). https://doi.org/10.1109/TCAD.2017.2752705
8. Cho, H., Cheng, E., Shepherd, T., Cher, C.Y., Mitra, S.: System-level effects of soft errors in uncore components. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **36**(9), 1497–1510 (2017). https://doi.org/10.1109/TCAD.2017.2651824
9. Cho, H., Mirkhani, S., Cher, C.Y., Abraham, J.A., Mitra, S.: Quantitative evaluation of soft error injection techniques for robust system design. In: Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, pp. 1–10. IEEE, Piscataway (2013)
10. Civera, P., Macchiarulo, L., Rebaudengo, M., Reorda, M.S., Violante, M.: An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. J. Electron. Testing **18**(3), 261–271 (2002). https://doi.org/10.1023/A:1015079004512
11. Dixit, A., Heald, R., Wood, A.: Trends from ten years of soft error experimentation. In: System Effects of Logic Soft Errors (SELSE) (2009)
12. Ebrahimi, M., Sayed, N., Rashvand, M., Tahoori, M.B.: Fault injection acceleration by architectural importance sampling. In: Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, CODES '15, pp. 212–219. IEEE Press, Piscataway (2015). http://dl.acm.org/citation.cfm?id=2830840.2830863
13. Entrena, L., Garcia-Valderas, M., Fernandez-Cardenal, R., Lindoso, A., Portela, M., Lopez-Ongil, C.: Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. IEEE Trans. Comput. **61**(3), 313–322 (2012). https://doi.org/10.1109/TC.2010.262
14. Espinosa, J., Hernandez, C., Abella, J., de Andres, D., Ruiz, J.C.: Analysis and RTL correlation of instruction set simulators for automotive microcontroller robustness verification. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2015). https://doi.org/10.1145/2744769.2744798
15. Fidalgo, A.V., Alves, G.R., Ferreira, J.M.: Real time fault injection using a modified debugging infrastructure. In: 12th IEEE International On-Line Testing Symposium (IOLTS'06), p. 6 (2006). https://doi.org/10.1109/IOLTS.2006.53
16. Foutris, N., Kaliorakis, M., Tselonis, S., Gizopoulos, D.: Versatile architecture-level fault injection framework for reliability evaluation: a first report. In: 2014 IEEE 20th International On-Line Testing Symposium (IOLTS), pp. 140–145. IEEE, Piscataway (2014)
17. Gaisler, A.: LEON3 processor. http://www.gaisler.com
18. Georgakoudis, G., Laguna, I., Nikolopoulos, D.S., Schulz, M.: REFINE: realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pp. 29:1–29:14. ACM, New York (2017). https://doi.org/10.1145/3126908.3126972
19. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N.D., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Simunic, T., Srivastava, M.B., Swanson, S., Sylvester, D.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**, 8–23 (2013)

20. Hari, S.K.S., Adve, S.V., Naeimi, H.: Low-cost program-level detectors for reducing silent data corruptions. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), pp. 1–12 (2012). https://doi.org/10.1109/DSN.2012.6263960
21. 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.J.: Design and architectures for dependable embedded systems. In: Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) pp. 69–78 (2011)
22. Henning, J.L.: SPEC CPU2000: measuring CPU performance in the new millennium. Computer **33**(7), 28–35 (2000). https://doi.org/10.1109/2.869367
23. Herkersdorf, A., Aliee, H., Engel, M., Glaß, M., Gimmler-Dumont, C., Henkel, J., Kleeberger, V.B., Kochte, M.A., Kühn, J.M., Mueller-Gritschneder, D., et al.: Resilience articulation point (RAP): cross-layer dependability modeling for nanometer system-on-chip resilience. Microelectron. Reliab. **54**(6), 1066–1074 (2014)
24. Höller, A., Macher, G., Rauter, T., Iber, J., Kreiner, C.: A virtual fault injection framework for reliability-aware software development. In: Dependable Systems and Networks Workshops, pp. 69–74 (2015)
25. Kaliorakis, M., Tselonis, S., Chatzidimitriou, A., Foutris, N., Gizopoulos, D.: Differential fault injection on microarchitectural simulators. In: 2015 IEEE International Symposium on Workload Characterization (IISWC), pp. 172–182. IEEE, Piscataway (2015)
26. Kanawati, G.A., Kanawati, N.A., Abraham, J.A.: FERRARI: a flexible software-based fault and error injection system. IEEE Transactions on Computers **44**(2), 248–260 (1995). https://doi.org/10.1109/12.364536
27. Kim, Y.M.: Early-life failures in digital logic circuits. Ph.D Dissertation, Stanford University (2013)
28. Kleeberger, V.B., Gimmler-Dumont, C., Weis, C., Herkersdorf, A., Mueller-Gritschneder, D., Nassif, S.R., Schlichtmann, U., Wehn, N.: A cross-layer technology-based study of how memory errors impact system resilience. IEEE Micro **33**(4), 46–55 (2013).https://doi.org/10.1109/MM.2013.67
29. Kleeberger, V.B., Mueller-Gritschneder, D., Schlichtmann, U.: Technology-aware system failure analysis in the presence of soft errors by mixture importance sampling. In: 2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS) (2013)
30. Lee, H., Song, Y., Shin, H.: SFIDA: a software implemented fault injection tool for distributed dependable applications. In: Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, vol. 1, pp. 410–415 (2000). https://doi.org/10.1109/HPC.2000.846589
31. Li, M.L., Ramachandran, P., et al.: Accurate microarchitecture-level fault modeling for studying hardware faults. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture (2009)
32. Li, Y.: Online self-test, diagnostics, and self-repair for robust system design. Ph.D Dissertation, Stanford University (2013)
33. Li, Y., Mutlu, O., Gardner, D.S., Mitra, S.: Concurrent autonomous self-test for uncore components in system-on-chips. In: 2010 28th VLSI Test Symposium (VTS), pp. 232–237 (2010). https://doi.org/10.1109/VTS.2010.5469571
34. Lorenz, D., Barke, M., Schlichtmann, U.: Monitoring of aging in integrated circuits by identifying possible critical paths. Microelectron. Reliab. **54**(6), 1075–1082 (2014)
35. Maniatakos, M., Karimi, N., Tirumurti, C., Jas, A., Makris, Y.: Instruction-level impact analysis of low-level faults in a modern microprocessor controller. IEEE Trans. Comput. **60**(9), 1260–1273 (2011). https://doi.org/10.1109/TC.2010.60
36. Meixner, A., Bauer, M.E., Sorin, D.: Argus: low-cost, comprehensive error detection in simple cores. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pp. 210–222 (2007). https://doi.org/10.1109/MICRO.2007.18

37. Mueller-Gritschneder, D., Dittrich, M., Weinzierl, J., Cheng, E., Mitra, S., Schlichtmann, U.: ETISS-ML: a multi-level instruction set simulator with RTL-level fault injection support for the evaluation of cross-layer resiliency techniques. In: 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 609–612 (2018)
38. Mueller-Gritschneder, D., Sharif, U., Schlichtmann, U.: Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator ETISS-ML. In: Proceedings of the International Conference on Computer-Aided Design, ICCAD '18, pp. 127:1–127:8. ACM, New York (2018). https://doi.org/10.1145/3240765.3243490
39. Oetjens, J.H., Bannow, N., Becker, M., Bringmann, O., Burger, A., Chaari, M., Chakraborty, S., Drechsler, R., Ecker, W., Grüttner, K., Kruse, T., Kuznik, C., Le, H.M., Mauderer, A., Müller, W., Müller-Gritschneder, D., Poppen, F., Post, H., Reiter, S., Rosenstiel, W., Roth, S., Schlichtmann, U., von Schwerin, A., Tabacaru, B.A., Viehl, A.: Safety evaluation of automotive electronics using virtual prototypes: state of the art and research challenges. In: Proceedings of the 51st Annual Design Automation Conference, DAC '14, pp. 113:1–113:6. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2593069.2602976
40. Openrisc 1000 architecture manual v1.1 (2014). http://opencores.org
41. Portela-Garcia, M., Lopez-Ongil, C., Valderas, M.G., Entrena, L.: Fault injection in modern microprocessors using on-chip debugging infrastructures. IEEE Trans. Dependable Secure Comput. **8**(2), 308–314 (2011). https://doi.org/10.1109/TDSC.2010.50
42. Pouchet, L.N., Yuki, T.: Polybench. https://sourceforge.net/projects/polybench/
43. Sanda, P.N., Kellington, J.W., Kudva, P., Kalla, R., McBeth, R.B., Ackaret, J., Lockwood, R., Schumann, J., Jones, C.R.: Soft-error resilience of the IBM POWER6 processor. IBM J. Res. Dev. **52**(3), 275–284 (2008). https://doi.org/10.1147/rd.523.0275
44. Stott, D.T., Ries, G., Hsueh, M.C., Iyer, R.K.: Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. IEEE Trans. Comput. **47**(1), 108–119 (1998). https://doi.org/10.1109/12.656094
45. Wang, N.J., Mahesri, A., et al.: Examining ace analysis reliability estimates using fault-injection. In: Proceedings of the 34th Annual International Symposium on Computer Architecture (2007)
46. Wang, N.J., Quek, J., Rafacz, T.M., Patel, S.J.: Characterizing the effects of transient faults on a high-performance processor pipeline. In: International Conference on Dependable Systems and Networks, 2004, pp. 61–70 (2004). https://doi.org/10.1109/DSN.2004.1311877

Online Test Strategies and Optimizations for Reliable Reconfigurable Architectures

Lars Bauer, Hongyan Zhang, Michael A. Kochte, Eric Schneider, Hans-Joachim Wunderlich, and Jörg Henkel

1 Introduction and Motivation

Runtime/reconfigurable architectures based on Field-Programmable Gate Arrays (FPGAs) are a promising augment to conventional processor architectures such as Central Processing Units (CPUs) and Graphic Processing Units (GPUs). Since the reconfigurable parts are typically manufactured in the latest technology, they may suffer from aging and environmentally induced dependability threats. In this chapter, strategic online test methods for dependable runtime-reconfigurable architectures as well as cross-layer optimizations for high reliability and lifetime are developed. Firstly, two orthogonal online tests are proposed that ensure reliable configuration of the reconfigurable fabric and aid fault detection. Secondly, a novel design method called module diversification is presented that enables self-repair of the system in case of faults caused by degradation effects as well as single-event upsets in the configuration. Thirdly, a novel stress-aware placement method is proposed that aims for slowing down system degradation by aging effects. The combined methods ensure reliable operation across architectural and gate level and allow to prolong the lifetime of dependable runtime-reconfigurable architectures.

The dependable operation of VLSI circuits is not only threatened by test escapes, intermittent or transient errors, but also by emerging hardware defects due to *aging* [11–13]. In nano-scale CMOS circuits, aging is related to *stress* which is defined as the condition under which a circuit structure experiences electrical and physical

L. Bauer (✉) · H. Zhang · J. Henkel
Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: lars.bauer@kit.edu; henkel@kit.edu

M. A. Kochte · E. Schneider · H.-J. Wunderlich
Institute of Computer Architecture and Computer Engineering, University of Stuttgart, Stuttgart, Germany
e-mail: schneiec@iti.uni-stuttgart.de; wu@informatik.uni-stuttgart.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_12

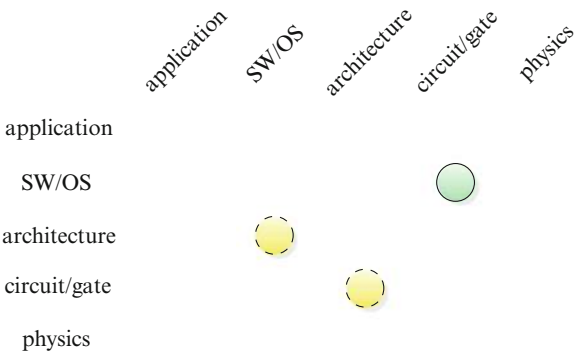


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

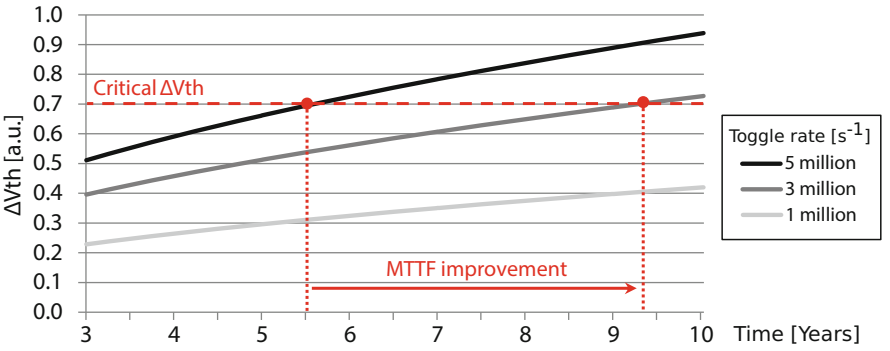


Fig. 2 Threshold voltage increase due to HCI-related stress (based on [22])

degradations. Two types of stress are distinguished: *static stress* and *dynamic stress*. Dynamic stress is typically characterized by the toggle rate of a transistor during which high currents flow between drain and source. A transistor is under static stress when an electric field is exerted across its gate oxide to induce a conducting channel. The stress is characterized by the duty cycle, i.e., the fraction of operation time the transistor is conducting. Dynamic stress leads to aging effects like Hot Carrier Injection (HCI), while static stress can lead to Bias Temperature Instability (BTI). Both are dominating aging mechanisms in nano-CMOS technologies [8, 16] and cause shifts in the threshold voltage ΔV_{th} of a transistor, which ultimately impacts the device performance over time. In this chapter, strategic online test methods for dependable runtime-reconfigurable architectures as well as cross-layer optimizations for high reliability and lifetime are developed (see Fig. 1).

The Mean Time to Failure (MTTF) of a transistor is defined as the time until its threshold voltage exceeds a certain critical value at which the transistor cannot deliver the required performance anymore. As shown in Fig. 2, the MTTF can be greatly increased if the transistor stress and consequently the threshold voltage shift are reduced.

Different aging models exist [2, 8], which indicate that both dynamic and static stress are generally *additive* through accumulation of the degradation effects. As a

result, this additive stress accumulation causes a *monotonic* increase in the transistor degradation over long terms. Although BTI degradation may experience a recovery effect, the recovery requires complex conditions or long relaxation periods [10] and will thus hardly affect the additive property. The monotonic and additive properties allow to consider stress during runtime (e.g., for resource management) with limited computational resources.

1.1 Application Model

In this work, a general application model is considered, as shown in Fig. 3. An application (Fig. 3a) consists of a mixture of normal operations, e.g., memory allocation and data preparation, and one or multiple computationally intensive parts, the so-called *kernels*. A kernel (Fig. 3b) corresponds to an outer loop that iterates through the whole data set and that contains one or multiple inner loops that work on small data parts, specified by the current iteration of the outer loop. For example, in a stencil operation of an image, the outer loop iterates over each output pixel and the inner loop computes the output value based on multiple neighboring input pixel values. Such an inner loop is a good candidate to be implemented as a *Special Instruction* (SI) that is composed of one or multiple *accelerators* of potentially different types. An SI (Fig. 3c) is represented by a data-flow graph (DFG) where each node corresponds to an accelerator and the edges correspond to data-flow between the accelerators [4]. Before the execution of an SI, all required accelerators need to be configured into the reconfigurable fabric, or otherwise the SI has to be emulated in software on the GPP. A sophisticated H.264 video encoder is the main application used for evaluation. The encoder consists of three kernels that require different SIs, implemented by nine types of accelerators [6].

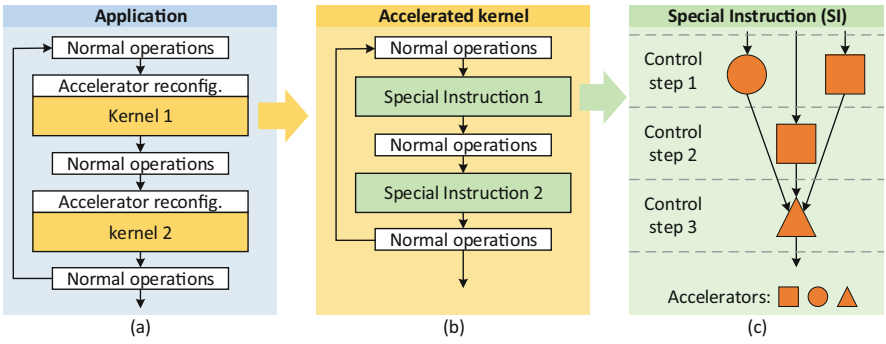


Fig. 3 This generic application model considers applications that consist of one or multiple kernels that may use Special Instructions (SIs) that are implemented by accelerators (based on [23])

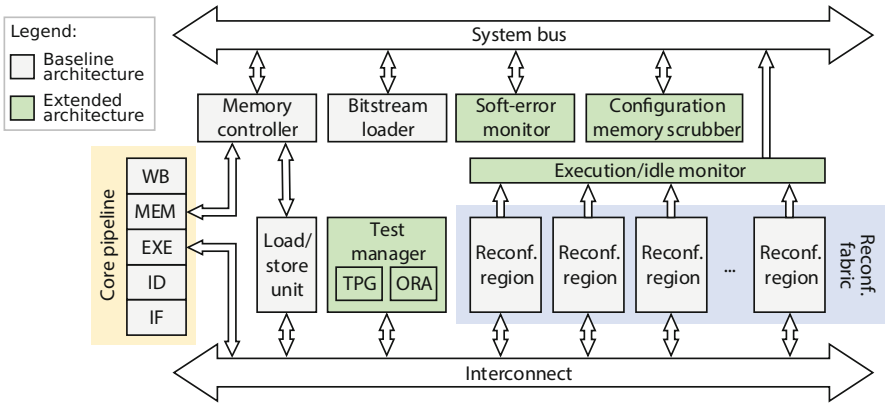


Fig. 4 Target reconfigurable architecture (based on [7])

1.2 Runtime-Reconfigurable Architectures

Runtime reconfiguration enables dynamic hardware customization to adapt to changing application requirements or environmental constraints, which maximizes performance at very low energy consumption. A reconfigurable architecture consists of a general-purpose processor and a *reconfigurable fabric*, partitioned into multiple *reconfigurable regions* (used to implement application-specific accelerators on-demand) that are interconnected via a communication infrastructure.

This chapter presents *Online Test Strategies for Reliable Reconfigurable Architectures* (OTERA), which targets FPGA-based fine-grained reconfigurable architectures as shown in Fig. 4. While transient faults due to single-event upsets are also addressed by OTERA (more details in [7]), this chapter focuses on aging-related challenges. To support dependable operation by online testing, stress balancing, and resource management for reliability and graceful degradation, a reconfigurable baseline architecture is extended by the following components:

- a test manager including a *test-pattern generator* (TPG) and an *output response analyzer* (ORA) to perform structural tests on the reconfigurable fabric and functional tests on the reconfigured accelerators;
- a workload monitor to track when a region is reconfigured and how often the currently configured accelerator is executed, which is used for stress estimation;
- a configuration memory scrubber to detect and correct errors in the configuration memory by periodical read-back and check of the configuration;
- a runtime system for dynamic dependability management by environmental monitoring, online test, reliability management, and aging mitigation.

The architecture is implemented using a LEON processor [9] and a parameterizable number of reconfigurable regions. A SystemC-based cycle-accurate simulator is used to evaluate the architecture and its runtime system. A hardware prototype is

developed on a Xilinx Virtex-5 FPGA and operates at a clock frequency of 100 MHz with a reconfiguration bandwidth of 50 MB/s.

FPGA hardware is composed of a two-dimensional array of reconfigurable primitive logic elements and routing structures that logic functions are mapped to. The two essential components are Configurable Logic Blocks (CLBs) and Programmable Switching Matrices (PSMs). The CLBs are the basic reconfigurable resources for implementing combinatorial and sequential logic functions. The interconnection between the components is configured using the PSMs. The logic function in a reconfigurable region is determined by configuration bits, called its *bitstream*, stored in SRAM-based configuration memory. Modern FPGAs support partial reconfiguration and allow to change the logic function without interrupting the operation in other parts of the chip [19].

An FPGA-based reconfigurable fabric, manufactured in latest technology nodes (e.g., 16 nm for Xilinx' UltraScale+ family), may suffer from degradation due to aging [10, 18]. Due to the increasing susceptibility of ever-shrinking nano-CMOS devices, these effects cannot be ignored anymore [11–13]. The resilience of the reconfigurable fabric is essential to the dependability of reconfigurable architectures, since most of the application's computations are offloaded to the fabric. The dependable operation of a hardware accelerator in the reconfigurable fabric relies on both the structural integrity of the fabric and the accelerator's functional correctness. While structural integrity of the reconfigurable fabric is a prerequisite for functional correctness of accelerators, the latter requires the correct completion of the reconfiguration process and correctness of the configuration data. However, the functionality of accelerators can be impacted during operation, for instance by SEUs that corrupt configuration data [7] as well as degradation of the hardware. To increase the dependability of the reconfigurable architecture, the structural integrity and functional correctness need to be addressed at different layers.

2 Fault Detection Through Strategic Online Testing

As latent defects and aging threaten the structural integrity of nano-CMOS devices, conventional manufacturing and burn-in tests are no longer sufficient to guarantee dependable operation over the whole lifetime. Therefore, *online tests* are required to check the system functionality. This task is particularly challenging for runtime-reconfigurable architectures, since the hardware organization changes during runtime as part of the normal operation [4]. This chapter presents two complementing types of online tests that are scheduled concurrently by the runtime system: *pre-configuration online tests (PRET)* and *post-configuration online tests (PORT)*.

2.1 Generation and Runtime Scheduling of Online Tests

PRET is designed to exhaustively test the underlying hardware structure in the reconfigurable fabric (e.g., logic resources in CLBs) periodically or on-demand. For PRET, an array-based structural test approach is used to generate *test configurations* for the exhaustive test of all logic resources in a reconfigurable region [1, 5]. Additional PRET test configurations are generated to target the application-dependent interconnects [6].

Since errors may also occur during the loading of bitstreams (e.g., due to faults in the configuration logic or transient events like SEUs), the configured function of the targeted region may be wrong or the configuration in other parts of the reconfigurable fabric may be adversely altered. For this reason, PORT is designed to perform at-speed functional tests on accelerators after their instantiation to ensure that they were configured correctly. At runtime, PORT also periodically checks the accelerators for malfunctions due to emergent permanent faults or soft errors in the configuration memory. An Automatic Test Pattern Generation (ATPG) tool is used to generate accelerator-specific test patterns to target the LUTs, combinational functions, and sequential elements in CLBs, as well as interconnects. The stuck-at fault model is used for components for which sufficient structural information is available to derive the faults and for the interconnects. For the remaining components, structural and cell faults are targeted during test generation resulting in a hybrid fault model [6].

Figure 5 shows the proposed online test flow for a reconfigurable fabric with three regions. In the first step (Fig. 5a), the runtime system decides that an accelerator shall be reconfigured into a particular region, which triggers the demand to test the hardware structures in that region before the actual configuration of accelerators (the so-called *on-demand PRET*). To exhaustively test all reconfigurable resources in the region, multiple *test configurations* (TCs) are required. The runtime system can choose to execute PRET incrementally to reduce the delay, applying only a subset of TCs (possibly none) prior to an accelerator reconfiguration. In practice, on-demand PRET-TCs are only scheduled after a certain number of *accelerator configurations* (ACs) have been configured. To reduce the impact on the application performance due to unavailable regions, PRET is only executed at times when the system needs to be reconfigured anyway. The runtime system tracks which TCs were applied to a region in the past and how much time passed since the last exhaustive PRET. Depending on this history, it activates PRET prior to an AC, reconfigures the selected TCs into the region, and uses TPG and ORA of the Test Manager to exercise the region (Fig. 5b).

In addition to on-demand PRETs, the runtime system also schedules *periodic* PRETs to ensure that seldom-reconfigured regions are properly tested. Note that PRET also needs to be executed regularly for regions that the application only reconfigures once and then never again (e.g., if the application only consists of one kernel; see Sect. 1.1). The reason is that PORT—despite its generally high fault coverage (see [6])—cannot always identify all faults. For instance, when an

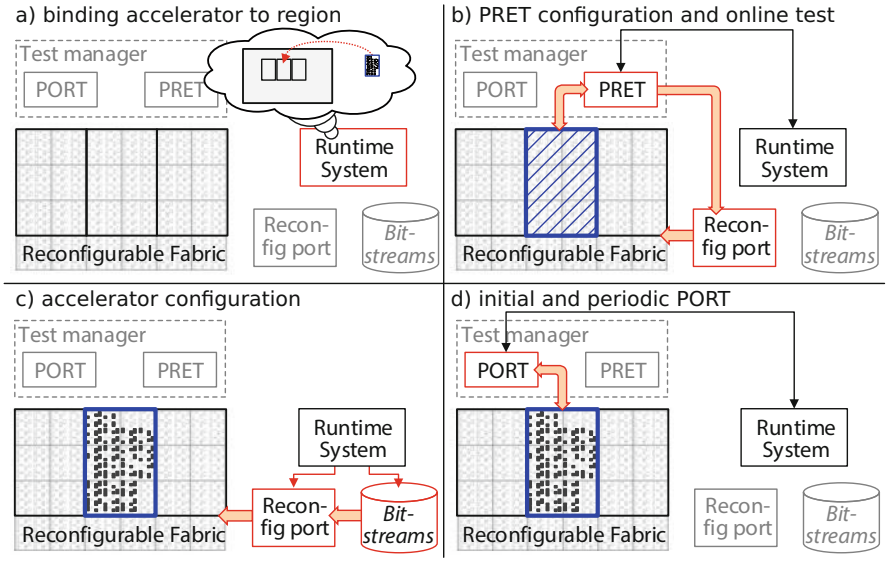


Fig. 5 Test flow with PRET and PORT (based on [6])

accelerator contains internal state, it is not always possible to apply an input value that propagates a possibly faulty value to an observable output. The periodic PRET is implemented using a timer interrupt and a handler that consists of two phases: (1) triggering the reconfiguration of a TC for a particular region and (2) executing PRET after the TC is reconfigured.

If no structural fault is found by PRET, the runtime system reconfigures the desired accelerator into the region (Fig. 5c). Before the accelerator is used by the application, the runtime system triggers an *on-demand PORT* (Fig. 5d) to test whether the reconfiguration process has completed without error. Additionally, accelerators instantiated in other regions are tested as well to check that they were not adversely affected by the reconfiguration. As PORT does not require any reconfiguration of TCs, it operates significantly faster than PRET and is also scheduled periodically during normal operation.

2.2 *Online Test Integration*

The test manager, TPG, and ORA are integrated into the reconfigurable architecture and coupled to the interconnect for the reconfigurable fabric such that communication channels between the regions and the test manager can be established. PRET and PORT are implemented as dedicated test-SI. In the base architecture, all SIs implicitly configure the interconnect infrastructure for the required data-flow

among accelerators and the system. The test-SIs reuse this mechanism to establish the connections between the test manager and the regions under test.

When the runtime system initiates a test-SI, the test parameters such as the target region or selection of test patterns are sent as the SI input data from the register file of the processor to the test manager. The test manager then generates the patterns by the TPG or sends stored patterns to the regions. While the PRET responses are sent back to the test manager for comparison, the PORT responses are compacted locally in space and time using a 32-bit multiple input signature register (MISR). The MISR is integrated into the interconnect infrastructure such that the outputs and the bus interface of a region are tested as well. After the test, the locally stored signatures are transferred to the test manager and compared with the expected signatures that are specific for each accelerator. At the end of PRET, the pass/fail information is written back to the register file of the processor. On-demand PORT is executed directly after an accelerator configuration to assure that the reconfiguration process completed without error and that the configured accelerator delivers the expected functionality. As PORT tests *all* configured accelerators in one test session, errors in the other accelerators, e.g., due to address decoder faults, are detected as well.

2.3 Experimental Evaluation

The effectiveness of PRET and PORT as well as the impact on the system performance is evaluated for the targeted platform. A test session consists of multiple test configurations (TCs) as shown in Table 1. In total nine TCs are required to test all logic primitives in the CLBs [1], and another nine TCs are required to test the interconnects of the accelerators of the H.264 application [6]. Each TC tests a subset of the logic primitives in the CLBs of a region or a set of interconnects used by the accelerator to be configured (Column 2). Columns 3 and 4 give the area overhead of PRET and the size of the generated partial bitstreams. The total area overhead introduced by PRET for all TCs is 17 CLBs. That is a one-time overhead to implement the test-pattern generator (TPG) and output response analyzer (ORA) for PRET, independent of which reconfigurable region is to be tested, whereas the other numbers in the table are per reconfigurable region. Note that the configuration time with tens of thousands of cycles dominates the actual application of the test patterns (Column 6).

The PRET overhead for the interconnect TCs is not applicable as the deterministic patterns are not generated by a TPG but stored similar to PORT patterns. The responses are compacted in the MISR introduced for PORT. In total 3780 bytes are required to store the test patterns of all interconnect TCs together with their signatures. The interconnect test reaches a fault coverage of up to 100% with the lowest being 98.28% [6].

The application performance loss introduced by PRET depends on the test frequency and number of reconfigurable regions. In this experiment, architectures

Table 1 Test configurations for CLBs and interconnects for reconfigurable regions of 4 × 20 CLBs (based on [6])

TC	Tested primitives	PRET overhead [CLBs]	Bitstream size [KB]	Freq. [MHz]	Test length [Patterns]
1	LUT conf. as XOR, connected to FF	2	24.0	207	64
2	LUT conf. as XNOR, connected to FF	2	24.0	207	64
3	Carry MUX, interleaved with MUX and latch	1	28.6	168	6
4	Carry MUX, interleaved with MUX and latch	1	26.1	154	6
5	Carry XOR, interleaved with MUX and FF	1	28.0	168	6
6	Carry XOR, interleaved with MUX and FF	1	28.2	154	6
7	Carry-in/-out with multiplexed scan chain	1	27.1	183	6
8	LUT conf. as SR with slice MUX	1	22.9	157	6
9	LUT conf. as RAM with slice output	7	22.3	225	320
10–18	Interconnect and PIPs of 9 accelerators	n.a.	29.6	78.8–191.9	13–123

with 5 and up to 14 reconfigurable regions are considered. The PRET handler is triggered every 1 ms and performs PRET if a region has not been tested for 500 ms. The observed test latencies until a region is completely tested ranged from 3.8 to 8.1 s, i.e., emergent faults do not remain undetected in the system for longer than 1.9 to 4.05 s on average. Table 2 reports the PORT performance impact and test latency. The upper part of the table shows the performance impact for PORT frequencies from 143 to 1000 Hz, i.e., test intervals from 1 to 7 ms. For each PORT frequency, the table shows the minimum and maximum performance loss of ten reconfigurable systems with different number of regions (5–14). The performance overhead due to PORT is very low (between 0.51% and 3.73%) and scales well with higher PORT frequencies. The observed worst case test latency, which corresponds to the longest untested time period of a region, is shown in the lower part of Table 2.

With PRET and PORT both enabled, the system is able to defend the configured accelerators against structural faults induced by aging effects or latent faults and transient events such as radiation [6]. For a PORT frequency of less than 100 Hz, the performance loss was dominated by the configuration frequency. After that point, the PORT frequency dominates the performance loss. The highest observed performance loss of only 4.4% occurs for a PORT frequency of 1000 Hz and a configuration frequency of 41 Hz.

Table 2 Performance loss and worst case test latency under PORT (based on [6])

		PORT application frequency [Hz]						
		143	167	200	250	333	500	1000
Performance loss	min.[a] [%]	0.51	0.59	0.72	0.89	1.20	1.81	3.68
	max.[a] [%]	0.56	0.63	0.75	0.92	1.23	1.85	3.73
Worst case test latency[b]	min.[a] [ms]	7.0	6.0	5.0	4.1	3.3	2.3	1.7
	max.[a] [ms]	7.8	6.8	5.8	4.8	3.8	2.8	1.8

[a]Summarizing ten reconfigurable systems with 5–14 regions
[b]Corresponds to the longest time period in the whole runtime in which a configured accelerator remains untested

3 Self-Repair by Module Diversification

Using PRET and PORT we can *detect* faults in the reconfigurable fabric. We now present a design method called *module diversification* [21] that generates a set of *diversified* configurations for each module/accelerator to *tolerate* any single-CLB fault and part of multi-CLB faults. The diversified configurations of an accelerator provide all the same functionality, but they vary in their CLB usage. They are reconfigured into the region at runtime without performance degradation. If a faulty CLB is detected, it is isolated from the system (i.e., a configuration is chosen that does not use it) to avoid any errors.

3.1 Diversified Configurations

A module defines the logic functions to be implemented in a region which consists of CLBs that are arranged regularly in a 2-dimensional array in the FPGA fabric. The CLB usage of a configuration is described by a *configuration matrix* as shown in Eq. (1) whose dimensions $X \times Y$ match the width X and height Y of a region in CLBs. If a configuration uses a certain CLB, the corresponding element in the matrix is 1, otherwise 0. For each module, a set $C = \{\mathbf{A_1}, \cdots, \mathbf{A_w}\}$ of configurations matrices with different CLB usage is generated. To be able to tolerate any single-CLB fault, this set of configurations must satisfy the *completeness condition* (Eq. (2)), which ensures that for any CLB in a region at least one diversified configuration $\mathbf{A_i}$ exists where the CLB is not used. Given that all diversified configurations implemented in a $X \times Y$ region occupy the same amount $U(< X \cdot Y)$ of CLBs (with at least one free CLB) a minimum number of w_{min} configurations (Eq. (3)) is required for the completeness condition [21].

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

$$\forall x, y, 1 \leq x \leq X, 1 \leq y \leq Y : \exists \mathbf{A_i} \in C \text{ with } [\mathbf{A_i}]_{x,y} = 0 \tag{2}$$

$$w_{min} := \left\lceil \frac{X \cdot Y}{X \cdot Y - U} \right\rceil \tag{3}$$

Two configurations $\mathbf{A}_i, \mathbf{A}_j \in C$ are said to be *maximally diversified* if their difference in the CLB usage is maximized. The *max diversification condition* [21] states that for every configuration $\mathbf{A_i} \in C$ there exists a maximally diversified configuration $\mathbf{A_j} \in C$ with a common number of CLBs:

$$\forall i, 1 \leq i \leq w_{min} : \exists \mathbf{A_j} \in C, j \neq i \text{ such that}$$
$$\sum_{x,y} \left([\mathbf{A_i}]_{xy} \cdot [\mathbf{A_j}]_{xy} \right) = \begin{cases} 2U - X \cdot Y & \text{if } \left(U > \frac{1}{2} X \cdot Y\right) \\ 0 & \text{else.} \end{cases} \tag{4}$$

3.2 Generation Algorithm

Algorithm 1 allows to generate maximally diversified configurations that satisfy the completeness condition [21]. Starting from an initial configuration $\mathbf{A_1}$ (Line 1) of a module, it incrementally generates diversified versions. A score matrix $\mathbf{G}$ stores

Algorithm 1 Generation of diversified configurations C

1. $C := \{\mathbf{A_1}\}$ // $\mathbf{A_1}$ is the initial configuration $(X \times Y)$
2. $\mathbf{G} := \mathbf{A_1}$ // Score matrix $\mathbf{G}$ stores swapping priority of CLBs $(X \times Y)$
3. $\mathbf{A_{new}} := \mathbf{A_1}$
4. **while** $|C| \neq$ desired number of config. $\wedge\ |C| \neq \binom{XY}{U}$ **do**
5. `zero_elem_list` $:= \{(x, y) \mid [\mathbf{A_{new}}]_{xy} = 0\}$ // unused CLBs
6. `cand_list` $:= \{(x, y) \mid [\mathbf{A_{new}}]_{xy} = 1\}$ // candidate list
7. sort `cand_list` in *descending* order according to the score in $\mathbf{G}_{xy}$
8. **for all** (x, y) in `zero_elem_list` **do**
9. `swap_candidates` $:= \{(p, q) \mid (p, q) \in$ `cand_list` and $\mathbf{G}_{pq} = \mathbf{G}_{\texttt{cand\_list[0]}}\}$ // all CLBs with the highest score
10. `farthest_swap_candidate` $:= (p, q) \in$ `swap_candidates` with max. Manhattan distance between (x, y) and (p, q)
11. `swap` $([\mathbf{A_{new}}]_{xy}, [\mathbf{A_{new}}]_{\texttt{farthest\_swap\_candidate}})$
12. `cand_list.pop(farthest_swap_candidate)`
13. **if** `cand_list` $= \emptyset$ **then**
14. **break**
15. **end if**
16. **end for**
17. **while** $\mathbf{A_{new}} \in C$ **do**
18. swap a random zero- with random one-element in $\mathbf{A_{new}}$
19. **end while**
20. $\mathbf{G} := \mathbf{G} + \mathbf{A_{new}}$ // update CLB score
21. $C := C \cup \{\mathbf{A_{new}}\}$
22. **end while**

for each CLB the number of available diversified configurations in C that use the respective CLB resources. The new configuration matrix $\mathbf{A_{new}}$ is initialized by $\mathbf{A_1}$ and modified in the inner loop (Lines 8–16) by swapping zero- and one-elements. The loop iterates over each element in $\mathbf{A_{new}}$ and swaps all zero-elements with one-elements in an order given by the score matrix (Line 7). If a CLB has a higher score, it is used more often in the diversified configurations. Thus the corresponding one-element in $\mathbf{A_{new}}$ will be swapped first. If CLBs have the same score, the distance-wise farthest one from the current zero-element is swapped first (Lines 9–11) so that the used CLBs are located near each other in the resulting configuration. The first w_{min} generated configurations correspond to the *minimal* set of configurations [21]. More configurations can be generated to achieve higher reliability or more alternatives during stress balancing (see Sect. 4). Random swapping in Line 18 allows to shuffle CLBs with different stress profiles. The algorithm terminates when either the desired number of configurations or all possible configurations have been generated.

3.3 Experimental Evaluation

To evaluate the reliability improvement and timing costs, the presented method is applied to a set of functional modules from the MCNC benchmark suite [20] and OpenCores.[1] The dimensions of the reconfigurable regions were chosen as 20 CLBs in height (80 CLBs for large modules) and 3–13 CLBs in width, which provides different degrees of CLB redundancy. For each module and region size the minimal set of configurations is generated using the proposed module diversification method. Since the design method applies additional constraints to prohibit certain CLB placements (`PROHIBIT` commands in Xilinx tools), additional routing effort is introduced that can affect the maximum clock frequency. To assess the impact on the system performance, the maximum frequency of diversified modules was compared to the original configuration. Initially, the clock frequencies of the modules ranged from 122.4 MHz (`apex2`) to 150.8 MHz (`pdc`). Experiments show that the timing penalty of the diversified configurations ranges from 0.04% (`aes_core`) to 9.7% (`misex3`). While the maximal frequency is given by the slowest configuration of a module, the original implementation also belongs to the configuration set and can be used when full performance is required. Also, if the system frequency is lower than the maximal frequency of the diversified modules, there are no timing penalties at all. Thus, module diversification is a promising approach to obtain fault tolerance without additional area overhead and little to no cost in system performance.

The *reliability* of an entity is the probability that the entity can operate without failure over a time period t. Without any fault-tolerance techniques applied, the overall reliability of a module with U CLBs depends on the reliability $R_{\mathrm{CLB}}(t)$ of each individual CLB (Eq. (5)). With module diversification, the reliability of the

[1] https://www.opencores.org.

module changes, as shown in Eq. (6). The first term states the probability that all CLBs are fault free. The second term aggregates all possible scenarios of multiple fault occurrences until all CLBs become faulty. The fault coverage $C_f \in [0, 1]$ is the fraction of f-CLB faults which are detected by an online test or concurrent error detection scheme such that reconfiguration with a diversified configuration allows to continue the operation. The fraction of f-CLB faults which can be tolerated with the set of available configurations is denoted by $\alpha_f \in [0, 1]$.

$$R_{\text{No\_FT}}(t) = (R_{\text{CLB}}(t))^U \tag{5}$$

$$R_{Div}(t) = R_{\text{CLB}}(t)^{XY} + \sum_{f=1}^{XY} \underbrace{C_f \alpha_f \binom{XY}{f} (1 - R_{\text{CLB}}(t))^f \, R_{\text{CLB}}(t)^{(XY-f)}}_{\text{Probability that } f\text{-fold CLB failures can be tolerated}} \tag{6}$$

We use the module `apex4` for the reliability analysis. Without fault-tolerance measures, the module has a very low reliability (≈0.91). Figure 6 shows the module reliability for a varying number of configurations and region sizes with CLB reliability $R_{\text{CLB}}(t) = 0.999$ and $C_f = 1.0$. The region size varies from 20×6 to 20×9 CLBs and corresponds to CLB redundancies from 22.4% to 111.8%. Larger region sizes reduce the overall module reliability since they have increased probability of a faulty CLB. By using diversified configurations, the module reliability increases dramatically. As shown, the tolerance of f-CLB faults rises with increasing number of configurations and very high module reliability is achieved (>0.999).

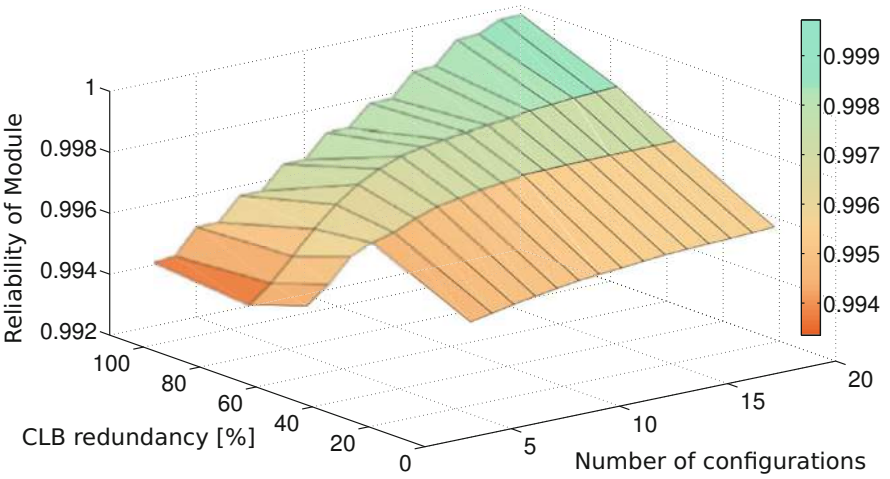


Fig. 6 Module reliability of `apex4` for different ratios of CLB redundancy and number of configurations with CLB reliability 0.999 (based on [21])

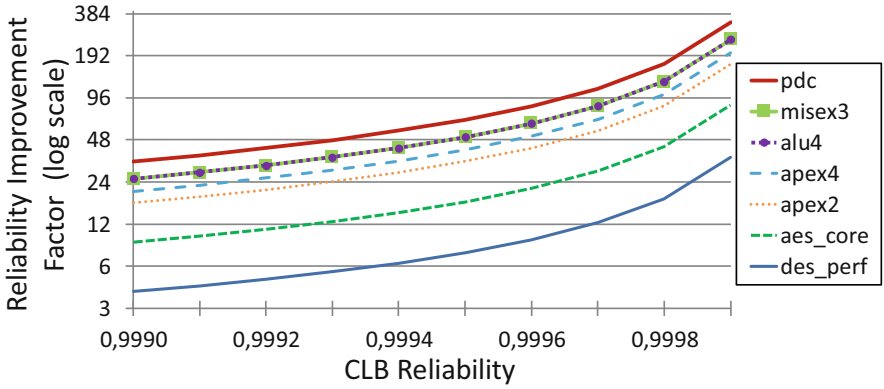


Fig. 7 Reliability improvement factor after module diversification (based on [21])

To estimate the effectiveness of the module diversification, the *reliability improvement factor* (RIF) is used [15]. The RIF is the ratio of the failure probability of the original system and the failure probability of the fault tolerant system using diversified module configurations (Eq. (7)). Figure 7 plots the RIF for the five investigated modules and CLB reliabilities ranging from 0.9990 to 0.9999. As shown, the proposed design method achieves reliability improvement factors of up to 330×.

$$\text{RIF} := \frac{1 - R_{\text{No FT}}}{1 - R_{Div}} \tag{7}$$

4 Prolonging Lifetime via Stress Balancing

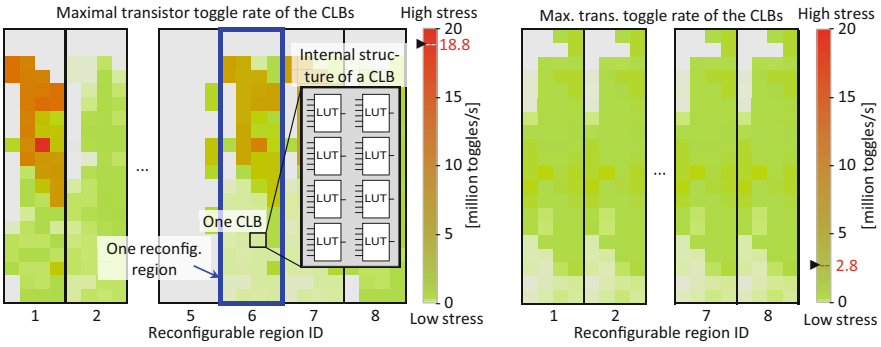
In addition to *reacting* on detected faulty CLBs (e.g., by using diversified modules as in Sect. 3), it is of crucial importance to *proactively* delay the occurrence of permanent faults (or increasing transistor switching delay) by aging mitigation via stress balancing. Different aging mechanisms have been reported for the current generation of CMOS designs, as discussed in Sect. 1. The main causes of these effects are environmental and electrical *stress*. Stress can be induced in different ways, e.g., through the presence of strong electrical fields or high current density [17, 18]. We propose the novel STRess-Aware Placement method STRAP that reduces the peak stress by aging mitigation. It combines complex offline optimizations at synthesis time with situation-dependent adaptation at runtime to optimize the intra- and inter-region stress distribution simultaneously. At runtime, STRAP places accelerators to different reconfigurable regions (i.e., it decides to which region they shall be reconfigured) while considering the induced intra- and inter-region stress distribution simultaneously. At synthesis time, STRAP diversifies stress during

place-and-route by preventing overlapping of high stress CLBs from different accelerators, which further improves the intra-region stress distribution at runtime.

4.1 Overview of the Stress-Aware Placement Method STRAP

The MTTF of a system is constrained by the component with the highest stress [17]. In order to prolong the MTTF of a reconfigurable fabric, stress accumulation on individual resources need to be avoid to reduce the peak stress. Figure 8a shows a typical reconfigurable fabric with 8 reconfigurable regions and 4 × 20 CLBs per region. The figure visualizes the distribution of HCI stress after running an H.264 video encoder. Higher HCI stress corresponds to more toggles per second of a transistor (see Sect. 1). For each CLB, the highest toggle rate of any transistor is identified and plotted in a color-scale from 0 (*low stress*, bright gray) to 20 million toggles per second (*high stress*, dark red). It is noticeable that several CLBs are not used (e.g., most parts of region 5), whereas some CLBs in region 1 contain transistors that are highly stressed. The latter represent *stress hotspots* where high stress accumulates in some of the components in the fabric which have a higher chance to fail much earlier than others, hence reducing the MTTF of the system.

The basic idea of STRAP is to place accelerators such that the *maximal* stress is minimized. Our *method* abstracts stress to the granularity of CLBs, whereas the *evaluation* of our method in Sect. 4.6 considers stress at transistor granularity. If the stress from a stress hotspot can be distributed to less stressed CLBs (like in region 5



(a) without stress balancing (b) with STRAP

Fig. 8 Transistor stress distribution in a reconf. fabric with eight regions; each region consists of 4 × 20 CLBs with 8 LUTs each (same setup as for evaluation); the color of a CLB corresponds to the highest toggle rate of any of its transistors; the symbol "filled triangle right" on the scale denotes the maximum stress over all regions (based on [22]). (**a**) Conventional execution without stress balancing. (**b**) Stress-aware placement in STRAP

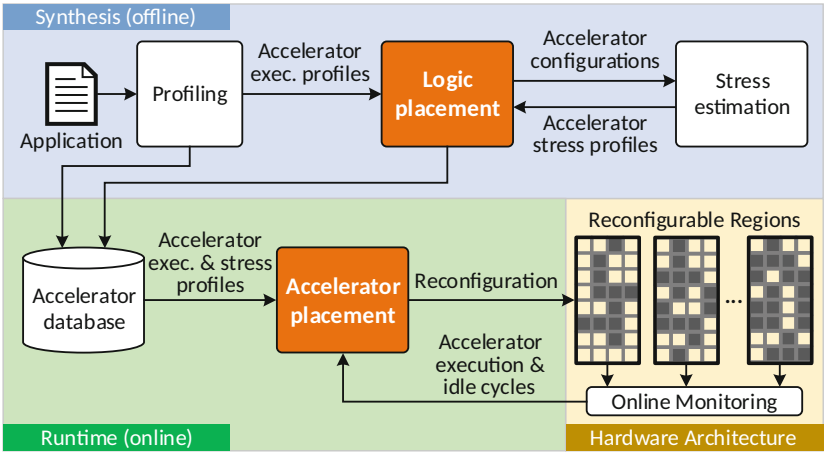


Fig. 9 Overview of the stress-aware placement method (based on [23])

in Fig. 8a), then the maximum stress in the reconfigurable regions is reduced (like in Fig. 8b), leading to increased MTTF.

Figure 9 provides an overview of the stress-aware placement method STRAP, showing the synthesis time techniques, the runtime techniques, and how they interact with the hardware architecture of a reconfigurable system. For logic placement at synthesis time, the challenge is to place-and-route accelerators in a way that supports stress balancing at runtime, but without having runtime information. STRAP first performs an offline application profiling of each application kernel to obtain estimates on (1) how often accelerators will be executed relative to each other and (2) how long each accelerator executes to finish its task. This information is used to steer runtime accelerator placement (Sect. 4.3) and synthesis time logic placement (Sect. 4.4).

Based on the accelerator configuration after place-and-route, the stress estimation process in Fig. 9 analyzes the signal activities in all CLBs used by the accelerator to obtain the information how much stress it induces to a reconfigurable region. Accelerator execution and stress profiles are stored together with the accelerator bitstreams in main memory for runtime decision making.

At runtime, STRAP decides into which reconfigurable region an accelerator shall be reconfigured, whenever the application demands different accelerators. It performs online monitoring of each region to track when the region was reconfigured last and how often the currently reconfigured accelerator was executed. Whenever a region is reconfigured, the execution counter and reconfiguration timestamp are read and reset. Together with the accelerator stress profile created at synthesis time, STRAP then calculates the exact *stress state* for all CLBs of the region. This information is used to decide the runtime accelerator placement.

4.2 Representation of Stress

Stress Granularity In order to handle the transistor stress in an algorithmic way, it needs to be represented compactly to allow an efficient runtime computation for the stress states of regions and the placement decision making. The transistors of a reconfigurable region are stressed by the reconfigured accelerator in a way that is determined by its logic functionality and input signal patterns. As the number of transistors in a region may be huge, the stress experienced by individual transistors is lumped to CLB granularity for the stress-aware placement method. *CLB stress* is defined as the sum of the stress experienced by all transistors in a CLB. With this definition, CLB stress preserves the additive property of transistor stress, i.e., the total stress a CLB experienced from different accelerators is the sum of the induced stress from individual accelerators.

Stress Accumulation With the established stress properties (see Sect. 1), the stress in the reconfigurable fabric can be described in a formal way. The stress state of a reconfigurable region (as it is visualized in Fig. 8) is denoted as matrix $\mathbf{S}$, where each entry represents the stress experienced by the corresponding CLB in the region. The stress that a particular accelerator induces per clock cycle is obtained from offline stress estimation and called *unit stress*, denoted by a matrix of the same size as $\mathbf{S}$. In general, the stress increase due to the work done by an accelerator is shown in Eq. (8). Matrices $\mathbf{s}^{\mathbf{unit}}_{\mathbf{exec}}$ and $\mathbf{s}^{\mathbf{unit}}_{\mathbf{idle}}$ denote the unit stress induced by the accelerator during execution or idle time and Sect. 4.6 explains how we use aging models to obtain these values by power/temperature analysis of placed-and-routed accelerators. Scalars τ_{exec} and τ_{idle} denote the number of clock cycles when the accelerator is executing or idle.

$$\mathbf{s} := \tau_{exec}\mathbf{s}^{\mathbf{unit}}_{\mathbf{exec}} + \tau_{idle}\mathbf{s}^{\mathbf{unit}}_{\mathbf{idle}} \tag{8}$$

The values for τ_{exec} and τ_{idle} are obtained from offline application profiling to construct the stress matrices (Eq. (8)) for every accelerator. The runtime system uses them to determine how much stress an accelerator would induce to a region *before* actually placing it. It also uses online monitoring (see Sect. 4.1) that provides the actual number of accelerator executions and idle times for each region *after* a computational kernel finished execution. This allows to keep track of the actual stress that a region experienced, which is the starting point for the next placement decision.

4.3 Runtime Accelerator Placement

The reconfigurable fabric consists of N equally sized rectangular regions. During runtime, the application requests to configure $M \leq N$ accelerators to speed up its computational kernels. The runtime system has to decide to which regions the M accelerators shall be configured, by first deciding which $N - M$ regions shall

not be reconfigured, e.g., by using a least recently used replacement policy. The decision to which of the remaining regions an accelerator is placed does not affect the application performance, but it affects the stress applied to the regions.

Each region contains $X \times Y$ CLBs with an (x, y) coordinate. The stress experienced so far by the CLBs in region k is denoted as $[\mathbf{S_k}]_{xy}$ and the stress that will be induced by an accelerator j is denoted as $[\mathbf{s_j}]_{xy}$ (see Eq. (8)). It depends on how often the accelerator will be executed, as determined by offline profiling (see Sect. 4.1). If an accelerator j is placed into region k, then the accelerator executions increase the stress state of the region to $\mathbf{S'_k} = \mathbf{S_k} + \mathbf{s_j}$. The challenge is to place each accelerator to a region, such that upon completion of the application kernel the maximum CLB stress over the N regions is minimized, i.e., $\max_{k,x,y} [\mathbf{S'_k}]_{xy}$ is minimized. It can be easily seen that the strict lower bound of the maximum CLB stress is given by Eq. (9), which is reached if and only if the stress is uniformly distributed over all CLBs. To achieve this at runtime, we propose a heuristic that follows these two rules: (1) maximal utilization of under-stressed CLBs within one region, i.e., the stress shall be evenly distributed among different CLBs within the region (*intra-region* distribution) and (2) avoid placing high stress accelerators into highly stressed regions, i.e. the stress shall be evenly distributed among different regions (*inter-region* distribution). The heuristic uses a profit function (Eq. (10)) for placing accelerator j into region k that considers the stress distribution within one region and across all regions, respectively.

$$\frac{1}{NXY}\left(\sum_{k}^{N}\sum_{x,y}[\mathbf{S_k}]_{xy} + \sum_{j}^{M}\sum_{x,y}[\mathbf{s_j}]_{xy}\right) \tag{9}$$

$$\text{Profit}_{jk} = \text{Profit}_{jk}^{intra} + \text{Profit}_{jk}^{inter} \tag{10}$$

To calculate $\text{Profit}_{jk}^{intra}$, the average CLB stress in region k is determined as AvgStress_k and then used to calculate the absolute deviation of the stress of CLB_{xy} in region k from AvgStress_k. The sum over all CLBs in region k denotes the intra-region stress imbalance. It is calculated (1) before placing accelerator j to region k and (2) after hypothetically placing it. The difference of these two values corresponds to the degree of increased stress imbalance if placing accelerator j to region k and is used as $\text{Profit}_{jk}^{intra}$. The idea for $\text{Profit}_{jk}^{inter}$ is very similar. There, the stress of region k is compared with the average stress of all regions before and after hypothetically placing accelerator j to region k [22].

The stress-aware runtime accelerator placement iterates over all required accelerators. In each iteration, it calculates the profits of placing the accelerator into all available regions and then places the accelerator into the region that provides the highest profit. The complexity of this algorithm is $\mathcal{O}\left(M^2XY\right)$. If the application does not reconfigure a region for a longer time, then this region would be constantly stressed by one accelerator without stress redistribution. As a solution, the runtime accelerator placement forces that region to be reconfigured after a user-defined time period that should not be too short to prevent increased reconfiguration overhead

and also not too long to avoid stress accumulation. For instance, a time period of 100 million cycles (1 s at 100 MHz) is short enough to avoid aging accumulation and the induced application performance degradation is only 0.21%.

4.4 Synthesis Time Logic Placement

Our runtime accelerator placement uniformly distributes the stress over all reconfigurable regions, compared to the stress-unaware placement. The maximal transistor toggle rate is reduced by more than 73% from 18.8 million toggles/s (see Fig. 8a) down to 5.0. However, when high stress CLBs of different accelerators *overlap* at the same relative (x, y) location, the runtime accelerator placement cannot achieve intra-region stress distribution. STRAP addresses this problem by applying placement constraints at *synthesis time* to diversify (similar to Sect. 3.1) the CLB usage among different accelerators, which reduces the overlapping of high stress CLBs. To minimize the timing impact on accelerators, STRAP only constrains which CLBs shall be used and leaves everything else to the vendor place-and-route algorithm.

The logic placement algorithm (Algorithm 2) diversifies the high stress CLBs of different accelerators to different CLB locations in the regions. First, unconstrained configurations of all accelerators are generated (Lines 1–5). For each accelerator

Algorithm 2 Stress-diversifying logic placement

Input: List of accelerators `Acc`.

1. **for** `j := 1 to len(Acc)` **do**
2. Place-and-route `Acc[j]` without any placement constraints
3. $\mathbf{s_j}$ `:= get_stress(Acc[j])`
4. `Acc[j].max_freq := get_max_freq(Acc[j])`
5. **end for**
6. `Acc := sort_ascending(Acc, key=max_freq)`
7. $\mathbf{R}$ `:=` $\mathbf{s_1}$
8. **for** `j := 2 to len(Acc)` **do**
9. `prohibit_xy := ∅`
10. **for** `x := 1 to Acc[j].n_cols` **do**
11. **for** `y := 1 to Acc[j].n_rows` **do**
12. **if** Condition Eq. (11) is satisfied for `(x,y)` **then**
13. `prohibit_xy.add((x,y))`
14. **end if**
15. **end for**
16. **end for**
17. Place-and-route `Acc[j]` with prohibited CLB locations listed in `prohibit_xy`
18. **if** Place-and-route failed **then**
19. `prohibit_xy.remove`$\left(argmin_{xy \in \texttt{prohibit\_xy}} \left\{ \left[\hat{\mathbf{R}} + \hat{\mathbf{s}}_\mathbf{j} \right]_{xy} \right\} \right)$
20. `goto Line 17`
21. **end if**
22. $\mathbf{R}$ `:=` $\mathbf{R}$ `+ get_stress(Acc[j])`
23. **end for**

configuration the CLB stress is estimated (see Sect. 4.2), and the maximal achievable frequency is extracted from the place-and-route log files (Lines 3–4). The generated initial configurations are then sorted in ascending order of their maximal achievable frequencies (Line 6). The fabric typically runs at the frequency of the slowest accelerator f_{min}. In order to minimize the impact on system performance, it is placed and routed without stress-diversifying placement constraints. Its CLB stress distribution is taken as the initial reference distribution (Line 7). As long as the proposed logic placement does not reduce the frequency of an accelerator below f_{min}, there is no performance impact/penalty for the whole system. During the generation of other accelerator configurations, $\mathbf{R}$ keeps track of the sum of the stress distribution of all $j-1$ previously generated accelerators, i.e., $\mathbf{R} = \sum_{i=1}^{j-1} \mathbf{s_i}$.

The remaining accelerators will be placed-and-routed again in ascending order of their maximal frequencies (Lines 8–23). To avoid that high stress CLBs of the currently placed accelerator `Acc[j]` overlap with those in previously placed accelerators `Acc[1],...,Acc[j-1]`, we prohibit the placement to specific CLB locations for `Acc[j]` (Lines 9–17) if Eq. (11) is satisfied, where L_j is the number of used CLBs by the currently place-and-routed accelerator `Acc[j]`. $\hat{\mathbf{R}}$ and $\hat{\mathbf{s}}_\mathbf{j}$ are normalized stress matrices of $\mathbf{R}$ and $\mathbf{s_j}$. In earlier iterations, the reference distribution is less even, which implies that few CLB locations in the reference distribution have much higher values than the others, and therefore it is less likely that the condition in Eq. (11) is satisfied. In turn, fewer locations are prohibited for placement in earlier iterations, which implies less timing impact on slower accelerators. If place-and-route fails due to too many prohibited CLB locations, the locations xy where the stress overlapping $[\hat{\mathbf{R}} + \hat{\mathbf{s}}_\mathbf{j}]_{xy}$ is lowest are removed from `prohibit_xy` (Line 19), and place-and-route is re-executed with the relaxed constraints.

$$\left[\hat{\mathbf{R}}\right]_{xy} > \frac{1}{L_j} \sum_{uv} [\hat{\mathbf{s}}_\mathbf{j}]_{uv}$$
$$\text{with } \hat{\mathbf{R}} = \frac{\mathbf{R}}{\max_{uv} [\mathbf{R}]_{uv}} \text{ and } \hat{\mathbf{s}}_\mathbf{j} = \frac{\mathbf{s_j}}{\max_{uv} \left[\mathbf{s_j}\right]_{uv}} \tag{11}$$

With synthesis time stress diversification, high stress CLBs from different accelerators are placed to different CLB locations, and thus better intra-region stress distribution can be achieved during runtime placement. After applying both stress-aware runtime placement and synthesis time stress diversification for dynamic stress, the maximal transistor toggle rate is further reduced by additional 44% from 5.0 million toggles/s down to 2.8 (see Fig. 8b).

4.5 Extended Accelerator Placement with Module Diversification

The module diversification method (see Sect. 3) generates a set of configurations for each accelerator that are diversified in terms of CLB usage. This not only allows to tolerate any single-CLB fault in a region but can also improve the stress distribution with the extra CLB diversity. When faults are detected in the reconfigurable fabric, the placement freedom of accelerators is reduced. The *placement freedom* of an accelerator corresponds to the number of regions for which the accelerator has at least one diversified configuration that can be placed into that region (i.e., that tolerates the permanent faults in that region). Such a region is called a *compatible region*. If the available regions (i.e., those into which no accelerators are placed by the placement algorithm so far) have rather many permanent faults, it can happen that no configuration of the accelerator can be placed into any of them. If an accelerator cannot be placed, then its hardware functionality has to be emulated in software on the processor pipeline, which comes at a significant performance loss.

To avoid such situations, the runtime accelerator placement (see Sect. 4.3) is modified to place the accelerators one after the other in ascending order of their number of compatible regions. If it comes to the situation that some accelerator cannot be placed into the available regions, then the algorithm re-evaluates some of its previous placement decisions (note that the actual reconfigurations are just started after all placements are finally decided). It tries whether it can *swap* one of the already placed accelerators into one of the still available regions such that accelerator can be placed into the region that became free due to swapping. When calculating the placement profit (see Eq. (10)), the algorithm also iterates through all diversified configurations to find out which configuration of the accelerator produces the highest placement profits.

4.6 Experimental Evaluation

For prototyping purposes, we have integrated STRAP into the Xilinx tool-chain and the runtime system of the target reconfigurable architecture. In our evaluation platform, each region consists of 4×20 CLBs with eight 6-input LUTs per CLB. STRAP performs optimizations on CLB granularity. To evaluate the actual stress for each transistor, a transistor-level model of LUTs using NMOS pass transistors for multiplexers is used [22]. To evaluate the threshold voltage shift due to stress, state-of-the-art aging models are employed (detailed equations and used parameters are given in [22]). The resource usage of each accelerator within one region for the H.264 application ranges from 8.8% to 66.3%. Our architectural simulator is used to evaluate the STRAP method for systems that differ in the number of reconfigurable regions and runtime strategies, and to compare it with related work.

Evaluation Flow The placed-and-routed accelerators are fed to Xilinx XPower analyzer to obtain the signal activities and power consumption of logic elements and nets. The power consumption is then aggregated to CLB granularity by summing up the power consumed by LUTs and their fan-in nets in one CLB. The leakage power of a region is proportional to its size. Architectural simulation produces the accelerator execution trace, i.e., the complete execution and idle history of each accelerator in each region. Together with the power profile of each accelerator, we obtain the power trace of each CLB. The power trace and the fabric floorplan of the FPGA[2] are then fed into Hotspot[3] [14] to obtain the temperature trace of each CLB, which will be used to evaluate the threshold voltage shift. The accelerator execution trace and the LUT signal activities of each accelerator are combined to calculate the LUT signal activities for the regions. This is then used to evaluate the stress of individual transistors by using the before-mentioned LUT transistor model.

The number of regions is varied from 5 to 12 and separate evaluation is performed for dynamic and static stress mitigation, since STRAP optimizes either for dynamic or for static stress. The baseline system does not use any stress distribution method. For comparison, two state-of-the-art stress distribution methods [3, 21] were implemented. Zhang et al. [21] use three different configurations for each accelerator and switch between them to migrate stress, whereas Angermeier et al. [3] consider the peak stress of regions to place an accelerator. As proposed for STRAP, Angermeier et al. [3] and Zhang et al. [21] were extended to replace an accelerator if its reconfigurable region has not been reconfigured for 100 million cycles (see Sect. 4.3). This improvement reduces the peak stress of [3, 21] and thus makes the comparison with state-of-the-art more competitive. Regarding temperature variation, a conservative comparison is performed. To calculate the threshold voltage shift for [3, 21], the lowest temperature that was observed for any CLB at any time in the obtained temperature trace is used as the constant temperature for all CLBs, while the highest observed temperature is applied for STRAP. Thus, the threshold voltage shift reported for [3, 21] is a lower limit, whereas the one for STRAP is a conservative upper limit.

Timing Overhead STRAP's stress-diversifying logic placement at synthesis time may affect the accelerator frequency. The place-and-route tool is given a target frequency of 250 MHz as timing constraint to obtain the maximum operating frequency of each accelerator. On average, the maximum accelerator frequency decreases by 7%. Since accelerators with longer critical path (lower maximum frequency) are imposed with fewer constraints (see Sect. 4.4), their maximum frequencies are less affected. The maximum *system* frequency is however limited by the accelerator with the longest critical path (in our case the PointFilter accelerator, which runs at $f_{min} = 89$ MHz). Therefore, STRAP has no negative timing impact on the system.

[2]Based on a high-resolution die image acquired from https://chipworks.com (now https://techinsights.com).

[3]Smallest possible heat spreader and heat sink with 10 µm thickness, ambient temperature 50 °C.

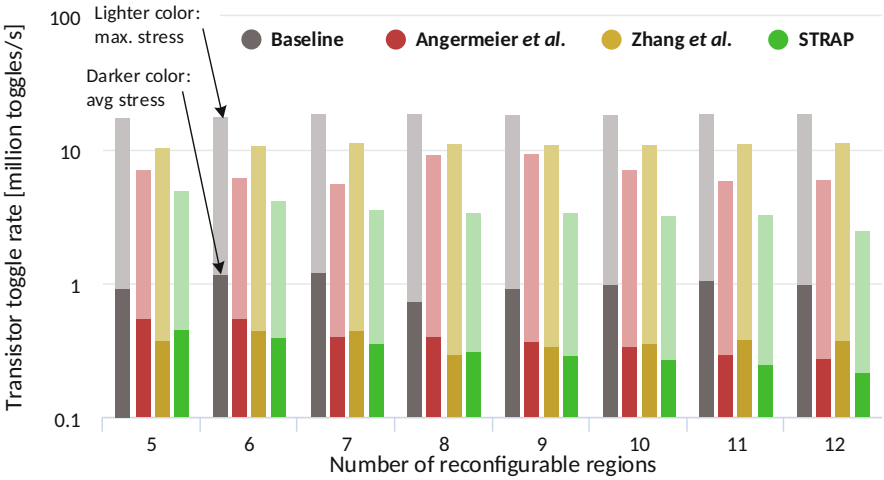


Fig. 10 The dynamic stress in systems with different number of reconfigurable regions when using our STRAP approach compared to the baseline, Angermeier et al. [3] and Zhang et al. [21] (based on [22])

Stress Reduction and MTTF Improvement Figure 10 shows the maximal (lighter color) and average (darker color; arithmetic mean) dynamic transistor stress, measured in million toggles/s, in the whole reconfigurable fabric for systems with different number of regions. It shows that all methods reduce the average stress compared to the baseline because they all distribute the stress to more transistors. While the reduction of the average stress is similar for all three methods, the reduction of the maximal stress (i.e., the critical part for system mean time to failure (MTTF)) differs significantly and requires both runtime and synthesis time optimization. The reason is that Angermeier et al. [3] perform only runtime inter-region stress distribution, while Zhang et al. [21] perform only synthesis time intra-region stress distribution for individual accelerators. In contrast, STRAP performs cross-layer stress-aware placement at runtime and synthesis time, which leads to the highest reduction of maximal stress in all evaluated cases. The reduction of the maximum stress by STRAP is up to 64% and 35% higher than the closest competitors w.r.t. dynamic and static stress, respectively. Table 3 summarizes the stress reduction.

Although during optimization only one type of stress is considered, actually both types of stress are reduced simultaneously. With STRAP targeting the static stress distribution, a reduction of 52% in dynamic and 38% in static stress is observed. When targeting dynamic stress, STRAP delivers 82% reduction in dynamic stress and 21% reduction in static stress. The reason behind the reduction of both stress types is that STRAP implicitly distributes the transistor usage as well, which reduces the individual static and dynamic transistor stress.

Table 3 Reduction of avg./max. stress and MTTF increase of STRAP and state-of-the-art [3, 21] compared to the baseline; averaged over all numbers of reconfigurable regions (based on [22])

	Reduction of avg. stress [%]		Reduction of max. stress [%]		MTTF improvement [%]	
Strategy	Dyn.	Stat.	Dyn.	Stat.	HCI	BTI
Angermeier et al. [3]	60.6	47.4	61.2	0.02	157.7	0.0
Zhang et al. [21]	62.6	49.6	39.9	4.5	66.4	2.3
STRAP	**67.9**	**59.6**	**80.5**	**33.1**	**413.0**	**13.4**

The MTTF improvement due to the stress reduction is calculated by assuming that a device fails when ΔV_{th} of any transistor exceeds 50% of its original value (V_{th0}). The MTTF improvement due to dynamic and static stress reduction is shown in the last two columns in Table 3. With the STRAP method, the MTTF improvement relative to the baseline is 413% and 13% in average for HCI and BTI aging, respectively. Relative to the closest competitors, STRAP achieves up to 177% and 14% MTTF improvement w.r.t. HCI and BTI aging, respectively.

5 Conclusion

The dependable operation of runtime-reconfigurable architectures is threatened by aging. This chapter presented novel methods to ensure reliable reconfiguration, mitigate aging, and tolerate emerging faults in the reconfigurable fabric. The *pre-configuration online tests* (PRET) and *post-configuration online tests* (PORT) check with minor application performance loss, if the reconfigurable fabric is faulty and if the reconfiguration process completed without errors during runtime. The *module diversification* design method generates the minimal number of diversified configurations required to tolerate at least any single CLB-fault in a reconfigurable region. The cross-layer stress-aware placement method STRAP mitigates aging by balancing stress both within a reconfigurable region as well as across all reconfigurable regions in the system. Relative to the closest competitors, STRAP achieves up to 177% and 14% MTTF improvement w.r.t. HCI and BTI aging. This shows that intelligently considering and managing aging threats during runtime can significantly improve the system dependability at limited overheads.

Acknowledgments This work is supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500—http://spp1500.itec.kit.edu).