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

© The Author(s) 2021

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_11

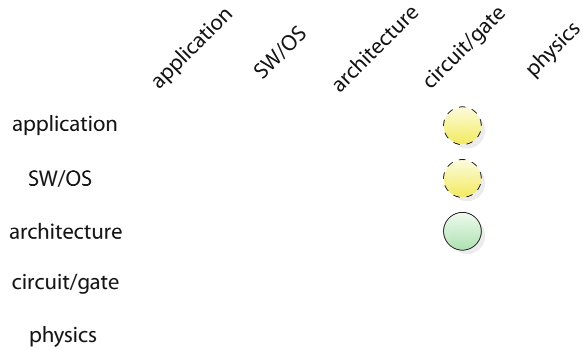

Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

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.

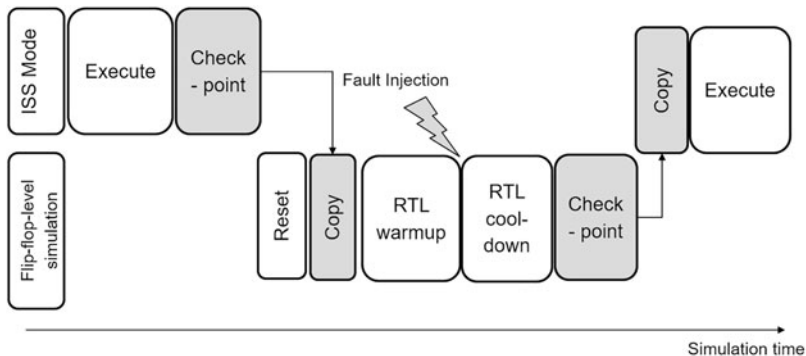

Fig. 2 Multi-level simulation flow of ETISS-ML

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.

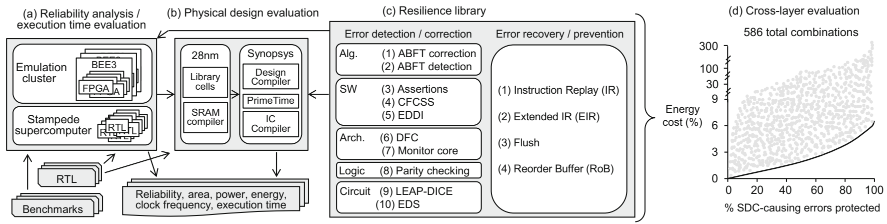

Fig. 3 CLEAR framework: (**a**) BEE3 emulation cluster/Stampede supercomputer injects over 9 million errors into two diverse processor architectures running 18 full-length application benchmarks. (**b**) Accurate physical design evaluation accounts for resilience overheads. (**c**) Comprehensive resilience 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 OoO	0%	0%	1.4%	1.4%	4.3×	1.2×	0%	N/A	1.01
	ABFT detection (without recovery—unconstrained)	InO OoO	0%	0%	24%	1–56.9%	3.5×	0.5×	0%	9.6M cycles	1.24

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

Resilience technique(s)	SDC improvement			
	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

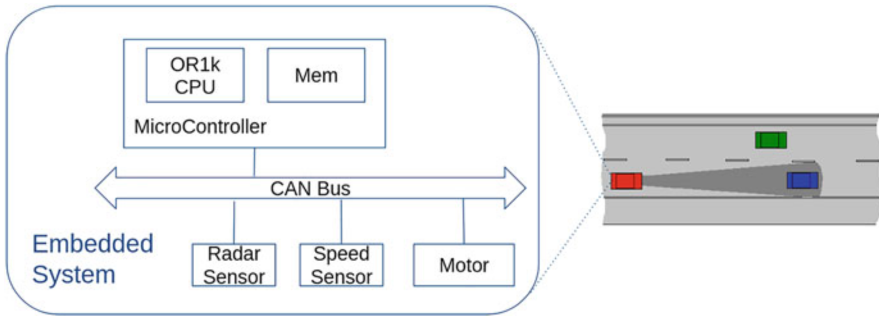

Fig. 4 SystemC VP of control system

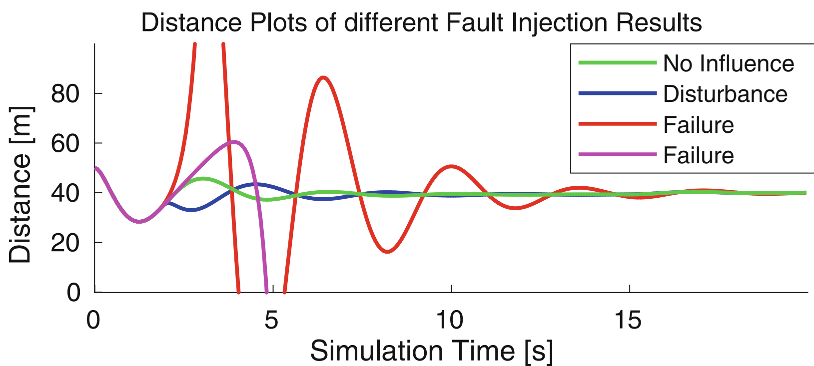

Fig. 5 Distance plotted for different FIs

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

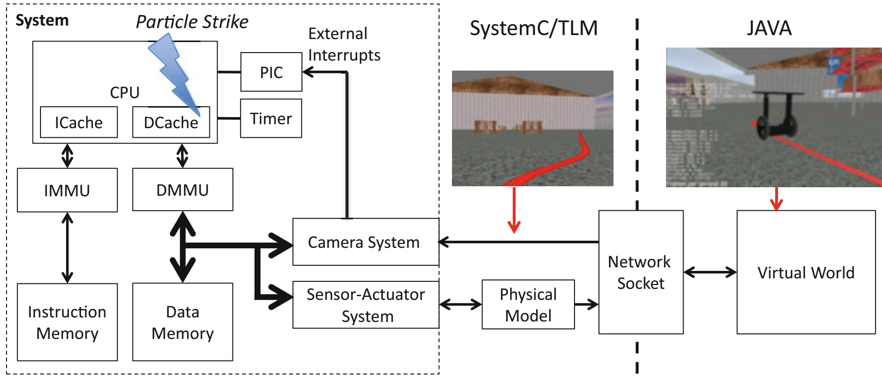

Fig. 6 Full simulation setup for self-balancing robot

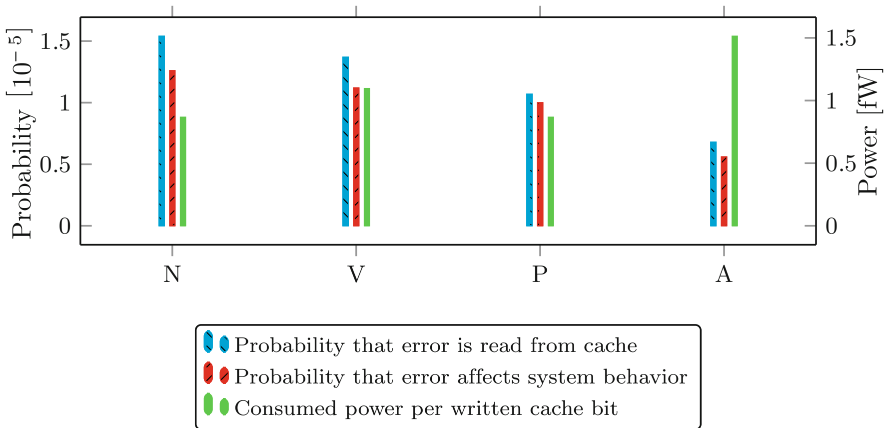

Fig. 7 Resilience exploration for cache of self-balancing robot

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

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

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