Challenges in Mitigating Soft Errors in Safety-critical Systems with COTS Microprocessors

Amer Kajmakovic*, Konrad Diwold*, Nermin Kajtazovic¶, Robert Zupanc¶

*Pro2Future GmbH & Institute of Technical Informatics, TU-Graz, Graz, AT
E-mail: (amer.kajmakovic, konrad.diwold)@pro2future.com
¶ Siemens AG, Graz, AT, E-mail: (nermin.kajtazovic, robert.zupanc)@siemens.com

***Abstract*—The number of Commercial-Off-The-Shelf (COTS) microprocessors and microcontrollers used in safety applications increased significantly over the last decade. In contrast to safety-certified microcontrollers, these microcontrollers are produced without integrated protection against memory soft errors, and limited in terms of available memory and computation power. However, due to the constant optimizations of the memory's physical size and the voltage margins, the probability that external factors, such as magnetic fields or cosmic rays, temporally alter a memory state (and thus cause a soft error) rises. Especially within safety-critical automation systems, it is crucial to address such errors and a wide range of error mitigation strategies have been proposed. In the context of established brownfield automation systems, the redesign and deployment of new hardware is usually not feasible. Therefore software-based strategies are required, which can be deployed on existing fail-safe architectures to further improve their performances, without requiring their rework or conceptual changes. This article identifies challenges associated with software-based soft error detection and correction strategies. Along with the challenges, a short overview of currently applicable software-based mitigation strategies is given and the strategies are evaluated.**

Keywords–soft errors; mixed-criticality; fail-safe; 1oo2D; embedded memory; hamming; parity bit; redundant parity.

I. INTRODUCTION

Given their ever-decreasing packaging size, semiconductors are becoming more susceptible to external influences such as alpha particles, cosmic rays or magnetic fields [1]. Figure 1 shows the correlation of error rates in semiconductors and technology/fabrication nodes (nm) size. It is noticeable, that the Soft Error Rate (SER) is increasing with decreasing node size, while the Hard Error Rate remains constant [2]. Given their low costs and good performances, Commercial-Off-The-Shelf (COTS) microcontrollers are increasingly deployed in safety applications [3]. In contrast to safety-certified microcontrollers, COTS microcontrollers are not produced with an integrated protection against soft errors. As a consequence, recent research proactively engages environmentally induced soft errors by developing new methods for error detection, mitigation, and data recovery [4]. This research direction has also yielded new challenges and requirements.

The importance of detecting and resolving soft errors is reflected by the numerous reports on soft error problems within safety-critical applications. Reports are coming from a wide range of industries, such as the automotive industry, space industry, or the medical industry. Duncan and Roche's analysis of semiconductor reliability in the context of autonomous driving [5] is devastating, as they conclude a (soft error induced) failure rate of 1 part per million per year. Given that a single-car implements approximately 8,000 semiconductors,

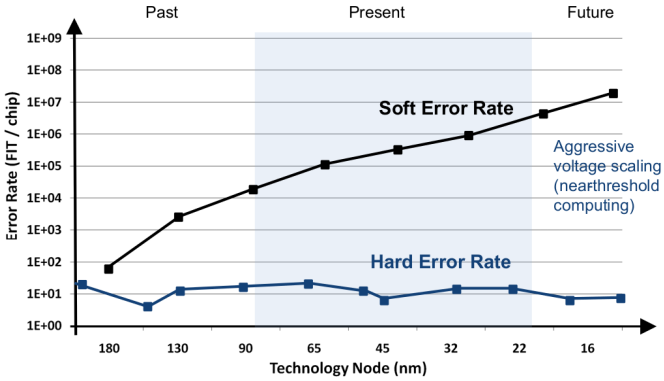

Figure 1. Software and hardware error rates in semiconductors [2].

the likelihood of a car exhibiting semiconductor induced errors within its lifespan (of 15 years) is around 12%. While the results of such failures are unclear while a car is operated, semiconductor-based soft errors can be resolved (fairly easy) by restarting the affected component. Not all safety-critical systems provide the luxury, of resolving an error by "turning it off and on again". Consider, for example, safety-critical nuclear power plant equipment, where restarting a device in the event of a soft error is not an option and could lead to fatalities.

Safety-critical applications usually exhibit different levels of criticality in terms of their underlying data. While a fraction of data is system critical (i.e., if affected by an error the consequences can be catastrophic), errors affecting non-critical data will not impact the safety of operation. This phenomenon is known as mixed-criticality. Incorporating mixed-criticality into the design of mitigation strategies, by devising and applying different detection and correction strategies on memory areas holding data of different levels of criticality, allows to further improve a system's availability while guaranteeing a correct treatment of system critical events.

The remainder of the paper is organized as follows: Section II presents an overview of the mitigation strategies. Section III defines the challenges and requirements for soft error software-based mitigation strategies in safety-critical applications. Section IV shows the evaluation of the two most suitable software-based mitigation strategies. Section V introduces how existing safety architectures can be improved with the software-based approach. In the last section, summary and future work of the paper are presented.

II. MITIGATING SOFT ERRORS

While soft errors represent the majority of memory errors, they can be prevented and/or corrected. To prevent soft er-

 ISBN: 978-1-61208-773-3

rors, memories require fault-tolerance. Fault-tolerance denotes a systems' ability to handle faults in individual hardware or software components, power failures, or other forms of unexpected problems, while still meeting its specification [6]. There are different approaches to achieving fault tolerance.

Shielding constitutes one of the first approaches, that made components fault-tolerant. Shielding is applied during the production phase, where a specific particle-resistant layer was deployed over the component's package. The layer reduces exposure of the bare component/device and prevents environmental particles from influencing under-layers of the package. Resistance to the electrical charges can also be achieved by using specific designs and materials for critical points in the component (e.g., strengthening the gate of the transistors). Although techniques used during the production phase have shown very effective against soft-errors, they always require additional materials and significantly increase the cost of the design and production.

If early stage design protections are not available, which is often true for the COTS microcontrollers, then the system redundancy is a very common solution to establish a fault-tolerant system. Four main types of redundancy exist: hardware, software, information, and time redundancy. The first three types of redundancy are achieved by providing additional components, functions, or data items that are not required for fault-free operation, but function as a backup for the event of a fault. Timing redundancy denotes repeating computations and a comparison of computational results from different timings.

1) Hardware redundancy: Hardware system architectures can provide fault tolerance via hardware redundancy. Safety-critical systems often adopt an N-modular (where $N > 2$) architecture, where the components exist in certain redundancy n and perform the same computations in parallel. The correct result is established based on majority voting. If one of the modules fails, the majority voter masks the fault by identifying the result of the remaining fault-free modules [6]. N-modular systems can yield towards a higher Safety Integrity Level (SIL), as they provide inherent fault tolerance and consequently a low failure rate. SIL is a quality indicator for systems that fulfill safety requirements in accordance with the IEC61508 standard. Many safety systems use simple architectures such as 1oo1D (1-out-of-1 with diagnostics) and 1oo2D (1-out-of-2 with diagnostics) [7]. In some cases, a diagnostic system is realized with an additional CPU (i.e., lock-step architecture) or with an additional watchdog (i.e., challenge-response architecture) [8]. These architectures are also known as fail-safe where in the event of a specific type of failure, the system inherently responds in a way that will cause no or minimal harm to equipment, environment or people. The main advantage is that these architectures have a good balance between functional safety (i.e., achieving high safety integrity) and development process costs. A shortcoming of hardware redundancy is its requirement for additional hardware. In the context of memory, it will increase its cost, weight, size, power consumption, and thus, impacts its designs and tests. Moreover, additional hardware needs to be in-calculated from the first stage of chip design. It is therefore almost impossible to upgrade already existing systems with additional hardware without degrading its performances, making it unsuitable for brownfield automation.

2) Software redundancy: Software fault-tolerant techniques are also based on the redundancy, which is applied to procedures, processes, data or the whole execution code. The most common type of software redundancy in embedded systems is the multiplication of data. A simple way of doing this is to store a variable copy simply transformed (e.g. with hamming distance 4 or simple inverse function) in a different memory area. This helps detect (via comparison), mitigate or recover corrupted data. The main disadvantage of software redundancy is memory consumption because multiplication of data, code or processes requires additional memory space that is usually limited in embedded systems. Also, in some cases, the code execution time could be significantly increased [1], [4].

3) Informational redundancy: The most prevalent type of redundancy in the context of memories is *Informational redundancy*. It assumes the addition of extra information to the data, which allows verifying the soundness of the information. Usually, this additional information are codes, which are computed based on the data itself. Those codes (so-called Error Detection And Correction codes (EDAC)) were firstly used in communication [6] for data recovery, but nowadays they are widely used in the memories [9]. The family of EDAC codes is expanding constantly, so far the most popular EDAC codes are: Parity Codes (error detection without recovery) [10], Hamming Codes (2-bit detection and 1-bit recovery) [10], Reed-Solomon and Bose-Chaudhuri-Hocquengham Codes (for multiple bits error masking) [9]. Also, some works considered the implementation of other EDAC codes used in communication such as: LDPC codes [11], RS codes, Turbo codes [12].

Most of nowadays EDAC codes for memories are implemented with an additional chip which is used to encode and decode EDAC codes [13]. These additional chips increase the cost of memory by 10-20%. Also, the memory's die-area increases by around 20% and processing speed decreases by 3-4% [9]. To avoid an increase in chip size and hardware redesigns, software-based EDAC codes have been proposed [14], [15]. However, such an approach leads to a decrease of available memory as well as an increase of computation time, access time, and the complexity of the overall system and usual trade-offs between listed parameters need to be made. It is worth mentioning that some conventional micro-controllers are already offering embedded memories with EDAC codes (i.e., hamming code or parity bit chips that can protect data from faults [16], [17]).

EDAC codes have two main properties that need to be considered: speed and quality. Speed is defined as the time needed for encoding/decoding EDAC codes and this time extends the overall memory access time, while quality can be determined as a number of the faulty bits that the code can detect and correct. Naturally, there is a trade-off between quality and speed. For higher quality, more complex EDAC codes are required, which allow correcting multiple bit-flips. In this case, both, code magnitude as well as computing demand are increased due to these adaptations. Faster and less memory expensive correction schemes are limited in terms of the number of bits that can be corrected.

Based on EDAC codes, a new method called scrubbing was also developed. The idea behind scrubbing is to periodically re-write data in its original location and eliminating soft errors, if they are correctable through EDAC [18], or copy of original data [19]. With this approach, an accumulation of soft errors

 ISBN: 978-1-61208-773-3

inside one region of memory can be avoided.

4) Timing redundancy: Another method that has been recently investigated is the so-called timing redundancy. It involves repeating a computation or data transmission two or more times and comparing results with previously-stored copies [6]. This type of redundancy is good when we need to distinguish between transient and permanent errors. If the fault is still there after repeating the test several times, then it's likely that error is a permanent one.

III. Challenges in mitigating soft errors

To overcome soft errors, and consequently lower their impact on the non-functional properties of the system, various methods for error detection, correction, and mitigation were introduced. Available methods can be distinguished into hardware- and software-based correction mechanisms. Hardware-based mechanisms provide error detection and correction on an architectural level and use specific hardware. As already stated, hardware approaches are not applicable in the brownfield i.e., existing devices or systems and they are usually demanding redesign and redeployment. For the already deployed systems or devices, software solution fits better because they can be deployed with a simple update or software patch and consequently costs are minimized. Software-based correction mechanisms operate on the memory itself without altering the underlying hardware or architecture. Depending on the application, an adequate correction quality is required, which denotes the fault magnitude a strategy is capable to detect, mitigate, and/or recover. Given that there is no such thing as a free lunch, soft error strategies require additional execution time and/or memory space, and therefore affect processor run-time and can cause memory overhead. In the next section, challenges for software-based solutions for soft-error mitigation will be discussed.

These observations lead to a general trade-off problem for the design and deployment of soft error detection and correction, as it is always required to balance the quality of detection (required by the underlying application) and the resources required to implement appropriate correction and detection strategies. Higher quality of the soft error correction will require more computation time, space, and sometimes additional hardware. Depending on the target system, this might lead to a violation of the system's requirements (in terms of cost, available memory space and computation time for the system's applications). In the following, the system's requirements are outlined in more detail.

1) Run-time performance: The development of methods, which provide sufficient error coverage, while keeping the impact on the system's run-time or memory overhead minimal is particularly important in the context of safety-critical systems. This is due to the fact that such systems have very strict timing requirements (i.e., norms in the field define specific timing limits here, such as Fault Tolerant Time Interval (FTTI) in ISO26262 or Process Safety Time (PST) in the IEC61508 standard). The FTTI constitutes the timespan between fault and hazard [20]. Faults must be detected and corrected within this interval. If a correction is not possible, the system must guarantee to reach a safe state within the FTTI. Therefore, the run-time performance of correction strategies plays a crucial role in the context of safety-critical systems as its application must not lead to a violation of the FTTI requirements.

2) Memory consumption: Many strategies require additional memory space for their implementation, which is used to store copies of data or code, or additional information (required by the method), such as Parity bits or Error Detection and Correction Codes (EDAC). From all software solutions, EDACs codes exhibit the smallest overhead because the ratio between additional bits required for protection and protected bits is always less than one while this is not the case for full redundancy. While in most cases EDAC codes can have a large memory footprint, parity bits constitute their most lightweight form. They allow monitoring the consistency of a memory region (with a defined length) based on a single bit, which denotes if the number of one-bits in the region is odd or even. With decreasing the size of the protected region this can lead to increased memory overhead. To give an example: the protection of 32bit word via hamming code will result in a 3.15% memory overhead). One-bit recovery of a 32-bit word, using Hamming code, would require additional 7 bits and thus result in a memory overhead of 22%.

3) Mitigation quality: The quality of a strategy is defined by its capability of detecting and correcting (recovering) faulty bits. Furthermore, detecting and correcting capabilities are expressed by the number of faulty bits that can be detected and corrected. The simplest EDAC code (Parity) can detect all odd bit flips but doesn't have recovering capabilities. On the other hand, a 2oo3 system can detect all bit flips and also can correct, but the complexity and consequently costs are higher.

In fail-safe systems, detection of an error is usually reflected with the safety feature because detection is enough to trigger activation of the safe state and prevents further safety issues. Between error detection and activation of a safe state, the system has a defined time for the recovery procedure. If recovery is not possible for any reason, the system will go into the safe state and availability will be affected.

4) Mixed criticality: When speaking about safety-critical memory one must distinguish between different levels of safety-criticality, which applies to the system data. Especially in safety-critical applications, some data may have a higher criticality level than the other. As already outlined, this phenomenon is known as "mixed-criticality". While adequate protection needs to be provided for the whole system, safety-critical data requires stronger protection. Several recent studies have investigated mixed-critically in memories, with a focus on data delivery and prioritization by data criticality [21].

Taking mixed-criticality into account when designing memory detection and correction strategies, allows enhancing the reliability and safety of the underlying system, as such strategies aim for increasing the protection of safety-critical memory parts. By treating different parts of the memory with a different criticality, the overhead of the correction strategies can be reduced (in contrast to the whole memory being subject to rigid correction/detection strategies). In addition, incorporating mixed-criticality can increase a system's availability, as faults in non-system critical memory areas will not necessarily lead to the halt of the system.

5) Memory organization: Because of the environmental changes, occurrences of soft errors in memory are not continuous. The chance of a cell being hit by an error is randomly distributed. Therefore, errors can appear at any time and in any type of memory or memory part. This can aggravate

 ISBN: 978-1-61208-773-3

memory protection and detection mechanisms as they are type-dependent. One can distinguish between two types of memory in embedded systems: non-volatile and volatile memory. Non-volatile memory sustains stored information during a loss of power (e.g., flash memory), while volatile memory needs constant power to retain stored data (e.g., SRAM) [22].

Embedded memory exhibits different regions: program memory, data memory, registers and I/O ports [23]. Also, from the software point of view, the memory layout of C/C++ programs consist of the different sections, that are saved in different memory regions. Typical memory representation of C/C++ programs consists of a code segment, data segment, uninitialized data segment (bss), stack and heap. All of this can impact the design of the correction/mitigation mechanism.

IV. Evaluation of EDAC codes

As shown in the last section, it is crucial to estimate the performance and overheads of soft-error mitigation strategies in order to identify those appropriate for one's problem domain and underlying system requirements. This section demonstrates how such an assessment could be performed, by calculating and comparing memory consumption and run-time performances of the Parity Bit (PB) and Extended Hamming Code (EHC).

The evaluation is performed for varying lengths of protected data (as strategies scale different with these). For the representation of the codes a common annotation (n, k) was used, where n denotes the number of total bits and k the number of the protected data bits. The number of required check bits can be easily calculated as $n - k$. Utilizing these parameters, memory consumption (mc) is calculated in (1) and exhibited on the Figure 2

$$mc[\%] = (n - k)/k \cdot 100\% \quad (1)$$

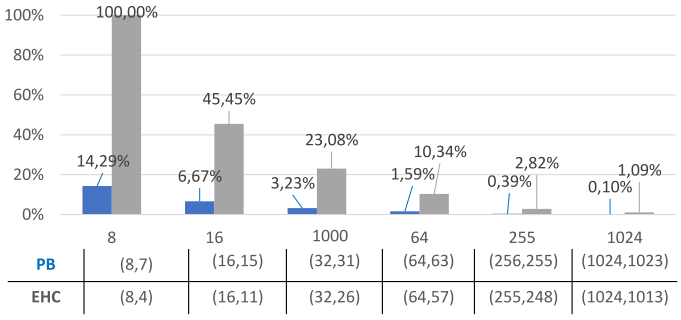

Figure 2. Memory overhead for different types of Parity Bit (PB) and Extended Hamming Code (EHC), where the x axis denotes total length of word and y denotes percentage of the memory overhead.

The run-time performance of a given strategy is closely connected with the complexity of the underlying algorithm. A good indicator of the algorithm's complexity is the number of logical XOR operators it requires for its implementation.

In the context of PB, a calculation stemming from [24] was used. The algorithm is based on the consecutive application of shift and XOR operators. Alternatively, a lookup table could be used to calculate the parity bits of 8-bit words. While using a look-up table will slightly increase the memory consumption of the algorithm it will decrease its complexity by 3 XORs.

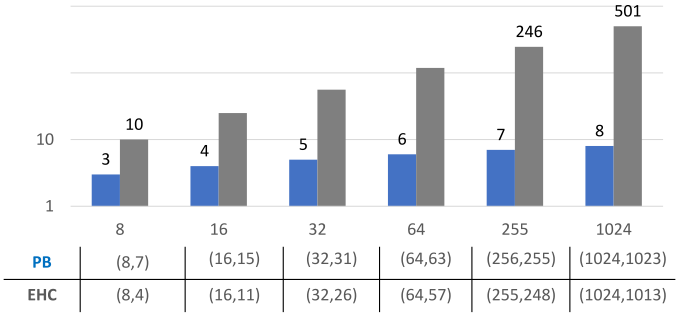

Figure 3. Number of XORs for encoding process for different types of PB(n, k) and EHC(n, k), where y-axis denotes the number of total XORs gates and x-axis the number of the protected data bits.

The number of the XORs for EHC was calculated according to (2).

$$XORs(k) = 2^{k+1} - k - 3 \quad (2)$$

where parameter k can be derived from the following form of hamming code annotation $H(2^k, 2^k - k - 1)$. The equation (2) stems from [24] where it was calculated for the EHC recursive encoding computation. Figure 3 shows the number of the XOR operator for varying lengths of protected bits.

In terms of mitigation quality, a big difference between PB and EHC can be observed. While PB is only capable to detect errors with an odd number of bit-flips (including single-bit errors), EHC can detect 2 errors and correct only one flipped bit. In the context of safety-critical systems, this low mitigation quality has a big impact on availability and safety.

In [25] a detailed report on the number of soft errors in SRAM memory (512K x 8-bit) is given, which were observed in space. Errors were recorded in a nanosatellite that was circulating the Earth's orbit. During the 2510 days of recording, four different types of 247593 soft errors occurred. The majority of these errors were single-bit errors (i.e. a total of 244150 errors constituting 98.6% of the recorded errors), while only 2996 errors (i.e., 1.21% of the recorded errors) were double-bit errors. Multiple bit (> 2) errors occurred at an even lower rate (corresponding to a total of 217 errors (0.08%)), while the remaining errors (230 (0.09%)) were classified as severe errors.

If the capability of presented algorithms was considered in this example in addition to considering the safety-critical scenario, PB would detect all single-bit errors and some of the multiple bit errors, leading to a detection rate of 98.75%. PB detection alone is not enough and would not increase the availability of the system, because without recovery the sole identification of an error would lead to the system being put into a safe-state as it is not safe to continue calculations. Using EHC, 99.8% error would be detected and 98.6% would be corrected. This means that the system's availability could be increased significantly as it would only be stopped (put in a fail-safe state) for 1.4% of the errors. This leads to the conclusion that (on its own) EHC is significantly better when it comes to safety and availability, however, this is also associated with the higher memory overhead and complexity (as shown before). Also, one should keep in mind that the SRAM used was relatively old (approximately 20 years old), and thus exhibits a lower probability for multiple bit errors because of the higher technology node. With newer memories (utilizing smaller technologies) the distribution of the error is

 ISBN: 978-1-61208-773-3

very likely to be different (i.e. more multiple-bit errors are to be expected).

In the context of safety-critical systems, the application of specific fail-safe architectures with hardware redundancy is very common. The next section will introduce a widely used fail-safe architecture and demonstrate how the application of simple EDAC codes can further improve its availability.

V. Enhancing an 1oo2D safety architecture

A typical representative of a fail-safe system is a 1oo2 architecture where the hardware, including sensor inputs, is independently implemented twice. This leads to a multi-core architecture similar to the one described in [26]. The output of these parallel lines is checked and selected by a voter [27]. Therefore, when the two outputs differ, the result leading to a safe and non-critical state is preferred and opted for by the voter.

From a memories' point of view, a 1oo2D architecture provides independent memories for each parallel line of computation. Two independent parallel memories ensure system hardware and software redundancy. This means that (besides memory specific data which is required for synchronization) identical data can be found on both memories (Figure 4 depicts the memory model in a 1oo2D architecture).

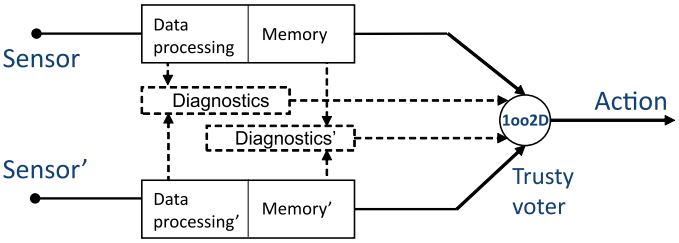

Figure 4. 1oo2D safety architecture.

All regions are equally exposed to the faults, however, different kinds of protection can be applied to different regions. Experts advise that protection should be implemented in the form of periodical tests run over data. As an exemplary guide, we can refer to the Safety manual [28] provided by STMicroelectronics for their micro-controllers. For soft errors, STMicroelectronics advises using redundancy for all safety-relevant variables. Usual solutions provide a copy of original data on the same memory chip or an additional (redundant) chip. The data is periodically compared with the original to detect the presence of errors [29]. When an error is detected, it is not clear which memory (or part of the memory) was affected, therefore such a solution leads to the detection but not to the correction and will result in the system transitioning into a safe-state.

A solution for overcoming this problem is to add mechanisms (on top of the existing architecture), which allow recovering faulty data and extend uptime of the system. Recovery mechanisms in this context are usually EDAC codes based. Adding additional hardware to the system is not feasible, as this would require redesigning the system from scratch. Another option is to apply software-based EDAC codes approaches.

Given that 1oo2D already provides the possibility to detect memory errors, the question arises on how existing architectures (i.e., 1oo2D) can be combined with software-based approaches.

A method, for enhancing existing 1oo2 hardware architecture, was proposed in our work [30]. This method constitutes an extension for mixed-critical real-time systems with an underlying 1oo2 architecture. We refer to it as Redundant Parity (RP). Figure 5 explains the basic concepts of the RP method. The method relies on 1oo2's ability to detect soft-errors and uses parity bits to establish the location of the error. Initially, the method generates parity bits for data that needs to be protected (i.e., data in redundant memories). When bit flips occur and 1oo2 comparator detects different bits in redundant data, the usual way is to generate a signal that will trigger the safe-state of the device. In contrast to that, the proposed method calculates new parity bits for both protected parts of the memories. In the next step, old parity bits are compared with newly calculated parity bits to establish the fault source. When the algorithm distinguishes between healthy and faulty data, the recovery phase is activated. Recovery is performed by simply copying healthy over the faulty data. To sum it up, the method uses 1oo2 architecture's inherent capability to detect bit flip in combination with parity bit to detect which of the redundant words is faulty and, in the end, it uses redundancy for recovery.

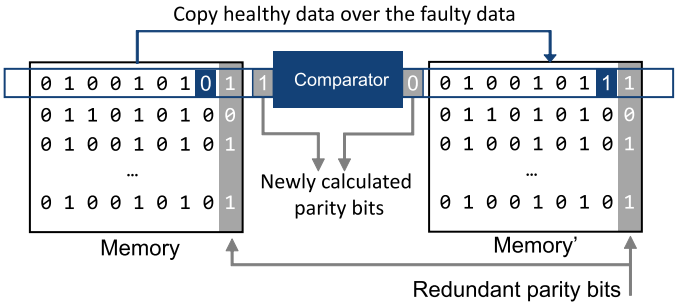

Figure 5. Redundant parity method.

The method allows for correcting single-bit soft errors (the majority of occurring soft-errors). In addition, odd multiple bits soft errors can be corrected and even multiple bits can be detected. In the context of the error data presented in the Section IV, this method would detect 100% of the errors and correct 99.4% of the errors. Memory overhead and complexity would be equal to double memory overhead and double complexity of the parity bit. Furthermore, the RP method provides separated detection and recovery phases, leading to less recovery time than in other EDAC methods. In addition, the proposed method is completely independent of the software architecture as it focuses on the memory's word level rather than on the variables or structures [31]. However, the results also show that the application of the approach is limited to a 1oo2 architecture, which already provides the required data redundancy as well as self-tests to detect errors in the data.

VI. Conclusion

The main goal of this work was to review software-based mitigation strategies for mixed-critical memories and identify challenges, that need to be considered. Soft errors, induced by external environmental factors, constitute a problem in memory operation. As safety certificated microcontrollers are expensive and complex industry is often utilizing COTS microcontroller.

To increase availability and reliability within COTS memories, a certain level of fault tolerance is required. Current

 ISBN: 978-1-61208-773-3

safety-critical applications rely on simple fail-safe architectures like 1oo1D or 1oo2D (which were outlined in Section V). The reliability and availability of fault-tolerant systems can be further improved if such simple fail-safe architectures are extended with software-based recovery techniques such as EDAC codes. In addition, deployment of the software-based EDAC codes does not require additional hardware or a redesign of the underlying architecture.

When deciding on a method to be implemented on existing hardware, one must be aware of the overhead costs, which are associated with a respective method, as it will likely increase run-time and/or reduce the available memory space. This aspect can be incorporated in strategy design, by directly addressing mixed-criticality of data within the correction and detection strategies, and differentiating among memory regions. The article tried to outline how such an assessment could be performed, by calculating and comparing memory consumption and run-time performances of different strategies, which can then be linked to the existing requirements of existing safety architectures, such as 1oo1D or 1oo2D.

The comparison of PB and EHC showed that, while PB exhibits less complexity and run-time overhead it will not increase availability per se, as detection will not lead to correction (in contrast to EHC). However, when PB is combined with existing 1oo2 safety architectures, a mitigation approach (named redundant parity) can be established, which is able to both detect and correct most of the soft-error occurring in memories, and thus significantly improve availability.

The method utilizes 1oo2's inherent capability of soft error detecting (achieved by a simple comparison test) and adds the mechanism of parity bits to distinguish between faulty and healthy data. In case an error is detected, the innate redundancy of the 1oo2 architecture is used to recover the error by copying healthy over faulty data.

ACKNOWLEDGMENT

The authors gratefully acknowledge the support of the Austrian Research Promotion Agency (FFG) (#6112792).

REFERENCES

[1] J. Vankeirsbilck, H. Hallez, and J. Boydens, "Soft error protection in safety critical embedded applications: An overview," in 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), November 2015.

[2] H. Iwashita, "International standards adopted by itu-t to address soft errors affecting telecommunication equipment," ITU-T International Telecommunication Union - Telecommunication Standardization Sector, Geneva, CH, Standard, 2018.

[3] H. Forsberg and K. Karlsson, "Cots cpu selection guidelines for safety-critical applications," in 2006 IEEE/AIAA 25TH Digital Avionics Systems Conference, Oct 2006.

[4] V. Thati, J. Vankeirsbilck, J. Boydens, and D. Pissoort, "Data error detection and recovery in embedded systems: a literature review," Advances in Science, Technology and Engineering Systems Journal, 2017.

[5] M. Duncan and P. Roche, "Paving the way towards autonomous driving — tackling soft errors to security challenges," in 2017 IEEE International Reliability Physics Symposium (IRPS), April 2017.

[6] D. Elena, Fault-Tolerant Design. KTH Royal Institute of Technology, Krista, Sweden: Springer, 2013.

[7] F. Handermann, "Process safety architecture system neutral solution comparison," Chemical Engineering Transactions, April 2016.

[8] R. Mariani and P. Fuhrmann, "Comparing fail-safe microcontroller architectures in light of iec 61508," in IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007), 2007.

[9] A. Mukati, "A survey of memory error correcting techniques for improved reliability," Journal of Network and Computer Applications, 2011.

[10] E. Fujiwara, Code Design for Dependable Systems: Theory and Practical Application. New York, NY, USA: Wiley-Interscience, 2006.

[11] S. Jeon, E. Hwang, B. V. K. V. Kumar, and M. K. Cheng, "Ldpc codes for memory systems with scrubbing," in 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, Dec 2010.

[12] B. Tahir, S. Schwarz, and M. Rupp, "Ber comparison between convolutional, turbo, ldpc, and polar codes," in 2017 24th International Conference on Telecommunications (ICT), May 2017.

[13] M. Restifo, P. Bernardi, S. De Luca, and A. Sansonetti, "On-line software-based self-test for ecc of embedded ram memories," in IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, October 2017.

[14] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-based ecc for gpus," Symposium on Application Accelerators in High Performance Computing, January 2009.

[15] D. Dopson, "Softecc: a system for software memory integrity checking," Ph.D. dissertation, Institute of Technology. Dept. of Electrical Engineering and Computer Science, Massachusetts, 2007.

[16] Intel® Embedded Memory User Guide, STMicroelectronics.

[17] MWCT101xS Safety Manual, NXP Semiconductors.

[18] G. Mayuga, Y. Yamato, T. Yoneda, M. Inoue, and Y. Sato, "An ecc-based memory architecture with online self-repair capabilities for reliability enhancement," in 20th IEEE European Test Symposium (ETS), 2015.

[19] R. Santos, S. Venkataraman, A. Das, and A. Kumar, "Criticality-aware scrubbing mechanism for sram-based fpgas," in 24th International Conference on Field Programmable Logic and Applications, 2014.

[20] IEC, "International Standard 61508 Functional safety: Safety related Systems," International Electrotechnical Commission, Geneva, CH, Standard, 2005.

[21] J. S. Miguel and N. E. Jerger, "Data criticality in network-on-chip design," in Proceedings of the 9th International Symposium on Networks-on-Chip, ser. NOCS '15. New York, NY, USA: ACM, 2015.

[22] K. Itoh, "Embedded memories: Progress and a look into the future," IEEE Design Test of Computers, January 2011.

[23] Reference manual for STM32 applications, Intel.

[24] L. Zhengrui, L. Sian-Jheng, and H. Honggang, "On the arithmetic complexities of hamming codes and hadamard codes," 2018.

[25] H. Caleb and B. Vipin, "Error detection and correction on-board nanosatellites using hamming codes," Journal of Electrical and Computer Engineering, 2019.

[26] F. Reichenbach and A. Wold, "Multi-core technology – next evolution step in safety critical systems for industrial applications?" in 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, September 2010.

[27] C. Preschern, N. Kajtazovic, and C. Kreiner, "Built-in security enhancements for the 1oo2 safety architecture," in 2012 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), May 2012.

[28] STM32F4 Series safety manual - user manual, STMicroelectronics.

[29] Handling of soft errors in STM32 applications, Intel.

[30] A. Kajmakovic, N. Kajtazovic, K. Diwold, R. Zupanc, and G. Macher, "Flexible soft error mitigation strategy for memories in mixed-critical systems," in 2019 ISSREW: International Workshop on Software Hardware Interaction Faults, Oct. 2019.

[31] A. Kajmakovic, R. Zupanc, S. Mayer, N. Kajtazovic, M. Höffernig, and H. Vogl, "Predictive fail-safe improving the safety of industrial environments through model-based analytics on hidden data sources," in Proceedings of the 13th IEEE International Symposium on Industrial Embedded Systems. IEEE Press, June 2018.

 ISBN: 978-1-61208-773-3