A Selective Soft Error Protection Method for COTS Processor-based Systems

Mohaddaseh Nikseresht[1], Jens Vankeirsbilck[1], Davy Pissoort[2] and Jeroen Boydens[1]
[1]Department of Computer Science, KU Leuven Bruges Campus
[2]Department Electrical Engineering, KU Leuven Bruges Campus
Spoorwegstraat 12, 8200 Brugge, Belgium
{ Mohaddaseh.Nikseresht, Jens.Vankeirsbilck, Davy.Pissoort, Jeroen.Boydens }@kuleuven.be

Abstract – **This paper proposes a framework that dynamically monitors a program's registers and analyzes their interaction with the memory at run-time. This information is used to implement the Selective-Swift-R fault tolerance technique, which enables us to selectively identify and protect vulnerable registers and achieve fault tolerance against soft errors. Being a pure software-implemented framework and error detection techniques, this research is suitable for commercial off-the-shelf processors. The evaluation based on number of Silent Data Corruptions and code size overhead has been evaluated for four different case studies compared to the unprotected program.**

Keywords – **Embedded Systems; Fault tolerance; Reliability; Soft Errors.**

I. Introduction

Over the past decade, advances in the microprocessor industry have enabled remarkable improvements in microprocessor performance and fabrication of high-density integrated circuits. Furthermore, due to their programmability, efficiency, and cost-effectiveness, Commercial Off-The-Shelf (COTS) electronic components have significant capabilities and benefits in the deployment of low-cost safety-critical and high-availability systems, in verity usage domain of embedded systems. However, these changes lead to the shrinking of transistor sizes and to increasing their vulnerability to radiation-induced single event effects (SEE). Such disturbances may be caused by energized particles in space or by secondary particles, such as alpha particles, generated by neutron-material interactions at ground level [1]. Among them, single event upsets or soft-errors have become one of the most challenging issues that limit the processor systems' reliability [2]. These non-destructive events can cause a microprocessor to generate a wrong result or lose the control flow in the system, all of which can have catastrophic consequences. This paper focuses on these soft errors. They are called soft errors or transient faults because these types of faults do not cause permanent damage and their impact can be minimized when the corrupted data is overwritten.

To address the issues caused by soft errors, certified microprocessors have typically used fine-grain redundant hardware. However, since COTS components prevent the use of hardware-based soft error reduction techniques directly inside the processor, other mechanisms must be used. The most popular coarse-grain hardware alternatives for these types of embedded systems are duplication or triplicate of COTS components [3]. Despite the fact that the majority of these hardware-based methods have an efficient response to transient faults, they have significant device disadvantages in terms of energy usage, die scale, construction time, and economic costs limiting their use in low-cost and portable systems.

As a cost-effective alternative to traditional hardware protection methods, in recent years, several proposals focused on redundant software, also known as Software Implemented Hardware Fault Tolerance (SIHFT). They are giving systems the ability to detect and correct faults [4-6]. Despite the technique's advantages, SIHFT strategies have substantial code and performance overheads, limiting their use. The increase in instructions and data may exceed the processor and memory resources, which are typically limited in embedded systems. Furthermore, the execution of redundant code increases application response time and can introduce unnecessary delays in real-time systems. Finally, execution overhead is a limiting factor for reliability in itself since it increases the time span over which processes are vulnerable to faults. To address these shortcomings, selective implementation of SIHFT has been suggested, with promising results on the design of dependable applications [7]. By protecting only a subset of elements e.g. processor registers, simple blocks of instructions, etc..., it is possible to achieve significant overhead reductions with minimal effects on overall reliability. The parts to protect should be selected based on their weakness or contribution to overhead.

Based on this concept, we propose a strategy to design fault-tolerant software effectively. The registers have been selected based on the method explained in SHARC [8]. According to SHARC, registers that are in direct touch with the memory are more sensitive than other registers. In light of that, this paper proposes a framework which dynamically monitors the registers used and ranks them based on their interaction with the memory. After selecting the most used registers, Selective-SWIFT-R is implemented to protect those registers and hence increase the reliability of the program [9].

The following describes the structure of the paper. Section 2 gives background for the relevant work. Section 3 presents the registers analysis framework and the implementation of Selective-SWIFT-R, Section 4 discusses the experiments and their results. Finally, Section 5 draws conclusions and makes recommendations for future work.

II. Related Work

In the design of digital circuits, hardware redundancy has traditionally been the most frequently applied technique to

978-1-6654-4518-4/21/$31.00 ©2021 IEEE

2021 XXX International Scientific Conference Electronics (ET) | 978-1-6654-4518-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ET52713.2021.9579862

resolve reliability difficulties. This comprises a wide range of solutions based on the following codes for detecting and correcting errors [10]. Recent solutions offer selective system hardening, which adds protection only to the most susceptible hardware portions [11], or decreasing performance impact by using partial redundant threading [12]. In general, these techniques have serious drawbacks for the system in terms of used resources, power consumption, die size, design time, and economic costs; limiting their use in low-cost and small systems. Due to these serious short comes a wide range of protective mechanisms based on the usage of redundant software have emerged in recent decades like SIHFT [13]. The vast majority of software-based solutions are designed to detect faults. Some utilize automated transformation rules to provide redundancy to high-level source code e.g., C language. In contrast, others use low-level instruction redundancy (assembly code) to reduce code complexity and output loss while also increasing detection rates [14-15]. Only a handful of these solutions have been extended to allow device recovery, but as a result, they have larger code sizes and longer execution times [16]. Overall, while software-based protection solutions can be a viable reliability option for low-cost COTS-based systems, they can also suffer from architectural concerns such as excessive memory overhead. This is especially true for solutions that attempt to not only identify but also recover from faults.

Designers are provided a wide range of alternatives, allowing them to go further into the software development area, considering factors such as code overhead, performance loss, and dependability level [17-18]. Ruano et al. suggest selective instruction replication which ensures application-level correctness in multimedia applications. In some situations, this type of application will accept a numerically incorrect execution, and the program output will always appear accurate to the user. Furthermore, recent hybrid hardware/software fault mitigation methods have shown encouraging results. These methods incorporate program replication as well as extra hardware resources [19–22]. However, in recent years, various approaches based on redundant software have been created, allowing programs to have both detection and fault correction capabilities. Because software-based solutions do not necessitate any changes to the microprocessor's hardware, they are particularly well suited for COTS-based systems that demand a high level of reliability. In this sense, our framework is proposes to analyze the most vulnerable registers in the program and selectively harden the code. To make the framework easy and practical to implement we have used Selective-Swift-R as the fault detection mechanism. Additionally, our framework described here can be employed as part of a more complex hybrid defensive plan or as part of a multi-layer defensive plan.

III. Implementation

This section begins with an explanation of the Register Analysis Framework, followed by an explanation of Selective-Swift-R.

A. Register Analysis Framework

Since memory is constructed to be as dense as possible, it is more vulnerable to ionizing particles than other circuit components. As a result, the registers that interplay with the memory are the first candidates to be hardened in a design. With this in mind, we have implemented a dynamic framework that analyzes and prioritizes the registers based on their interaction with the memory. And later we protect the register with Selective -R method; figure 1 shows the inner working of our register analyzer framework. The process starts by initializing the register Usage Table to all zero, indicating no register has interacted with memory yet. Next, the target program is executed, one step at the time. For each target program step, the framework fetches the instruction from the processor. The fetched instruction, saved in the **`currInstr`** variable, is then analysed for its type. If it is of type **`load`**, the registerUsageTable is updated, increasing the count for the register containing the memory address the data is loaded from. If currInstr is not of type **`load`** but of type **`store`**, the registerUsageTable is updated, increasing the count for both the register containing the memory address and for the register containing the value to be stored to memory. Finally, a check is performed to see whether or not the current instruction is the last instruction in the target program. If this is the case, the framework stops executing and returns the registerUsageTable. In the other case, the next step of the program is executed and the analysis is repeated until the end of the program is reached. The returned registerUsageTable contains the number of interactions each register had with memory. This forms the base for selecting the most vulnerable register(s). In this paper, the register with the highest number of interactions with memory is selected to be protected.

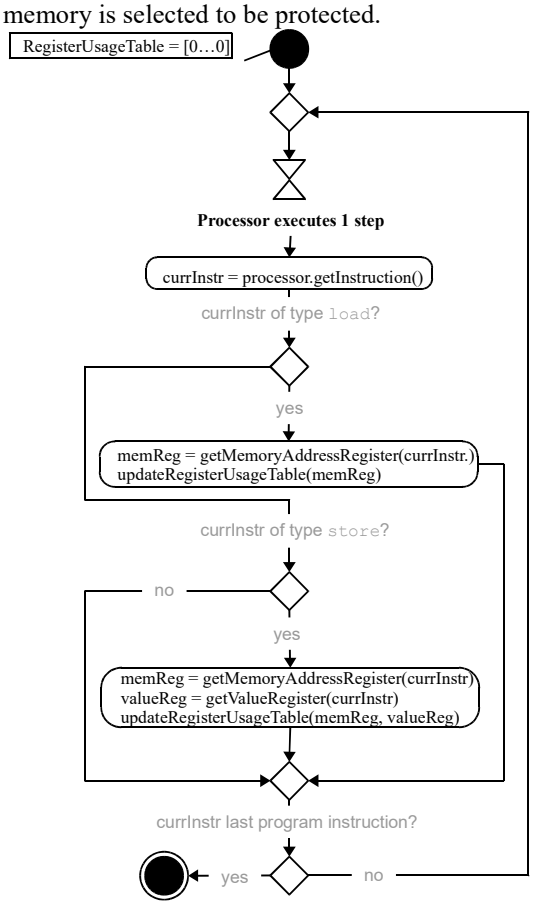

Fig. 1. The Register Analyzer Framework

B. Selective-SWIFT-R

To protect a register, a fault tolerance technique that only covers a subset of elements is needed. To do so, we concentrate on using the Selective SWIFT- R methodology suggested by Felipe et al. [9]. It is a software-only recovery strategy based on low-level instruction transformation rules and the well-known Triple Modular Redundancy (TMR). It joins three copies of the program together and places majority voting before vulnerable instructions. To put it another way, it comprises the duplication of data and instructions and the addition of data consistency checking points, through majority voters. Each secure register needs two additional copies. This means that there will be a total of $2 * n$ new registers where n is the number of registers we want to secure. The application of Selective SWIFT-R to source code (assembly code) is explained in Algorithm 1. To clarify the process more, the following example of Fig. 2 is provided. This example has been taken directly from the Selective SWIFT-R paper. In Fig. 2 the difference between Non-hardened, Selective SWIFT-R and SWIFT-R is shown. In non-hardened code, no register is protected. However, on the second and third case only one register is protected; thus, this is a good representation for Selective SWIFT-R when we partially protect one register. Each register has two shadow registers in each case. In the last table, both registers of the program are protected, representing the implementation of SWIFT-R. The number of registers has been double in this case, increasing the imposed overhead.

IV. EVALUATION

To assess our proposed approach, fault injection was used to determine the fault tolerance capabilities of Selective SWIFT-R. This section first discusses the setup for the experiments and the used case studies. Next, the measured criteria are explained. Finally, the experiment results are shown and discussed.

Setup for the experiment:

The Imperas simulator was used to create the framework [24]. The Imperas simulator is an instruction set simulator that allows the target instructions to be executed at the speed of the host. ARM, MIPS, PowerPC, and a variety of additional platforms are supported by the Imperas simulator. Furthermore, we employed a simulated target for the fault injection because prior tests showed that utilizing a real target results in communication cost and failures. The communication overhead is caused by utilizing USB or JTAG to interact with the target, and the failures are caused by the on-chip debugger failing to alter the CPU registers. These drawbacks were overcome by moving to a simulated target. In our earlier articles, we discussed the implementation specifics of the data flow error injection processor [25].

Case studies:

The selected case studies are an implementation of the following four algorithms:

• **Bit Count (BC):** Bit Count (BC): The bit count algorithm counts the bits set, i.e. the 1's, in the given data word, also known as the hamming weight. This functionality is, amongst others, used in the communication domain to calculate a parity bit or in the cryptographic domain to calculate keys [26].

• **Bubble Sort (BS) and Quicksort (QS):** The bubble sort and quick sort algorithms were selected because the sorting of information is used in different systems, e.g., to assign priorities or to enable faster analysis of data [27].

• **Matrix Multiplication (MM):** The matrix multiplication algorithm was selected because matrices and matrix multiplication are used in many different embedded domains such as image processing, e.g., CAT and MRI scan, robotics and data compression [28].

The case studies were run on a simulated version of an ARM Cortex-M3 CPU, which is a popular 32-bit processor used in a variety of embedded applications.

Criteria:

We used two criteria to compare the protected and unprotected versions of case studies: silent data corruption and code size overhead [25].

• **Silent Data Corruption (SDC):** we determined the effect of each injected fault. SDCs are the faults that were not detected by the implemented technique and caused the algorithm to produce a wrong result. They are the most critical type of error and can damage the system That's the reason we focus on them in this paper.

• **Code Size Overhead:** data flow error detection approaches that are implemented in software add extra instructions to the code they need to protect. This signifies that the pro-

		Selective SWIFT-R		SWIFT-R
#	Non-Hardened	Protected register: s0	Protected register: s1	Protected registers: s0, s1
1	LOAD s0, 00	LOAD s0, 00	LOAD s0, 00	LOAD s0, 00
2		*create s0 copies*		*create s0 copies*
3	LOAD s1, 2A	LOAD s1, 2A	LOAD s1, 2A	LOAD s1, 2A
4			*create s1 copies*	*create s1 copies*
5			*majority voter for s1*	
6	ADD s0, s1	ADD s0, s1	ADD s0, s1	ADD s0, s1
7		*ADD s0', s1*		
8		*ADD s0", s1*		
9		*majority voter for s0*		
10			*majority voter for s1*	*majority voter for s1*
11	STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)

Fig. 2. The Difference between Non- hardened, Selective SWIFT-R and SWIFT-R

tected code has a larger code size than the unprotected code. We measured its code size and compared the results to the unprotected code. The code size is measured by counting the number of assembly code instructions in the case study:

$$code\ size\ rel.\ to\ unprotected = \frac{Instruction\ Count\ Protected}{Instruction\ Count\ UnProtected}$$

For each of the selected case studies, we created five datasets and ran the framework on each of them to determine which register was the most vulnerable. The result of the execute in BS dataset is presented in Fig. 3, as it can be seen R3 is the most vulnerable registers in BS case study. The outcome of the register analysis for all four case studies is summarized in Table 1, which indicates which register is deemed most vulnerable for each case study.

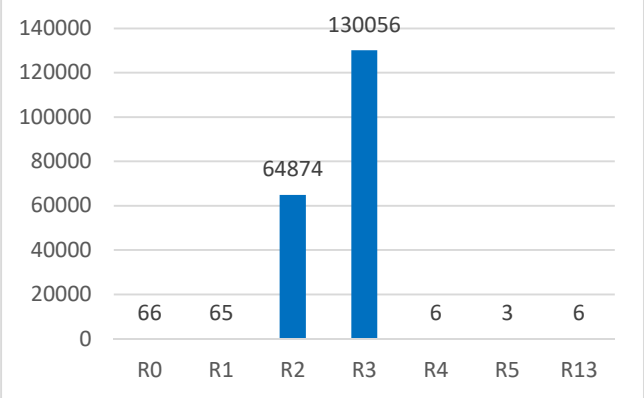

Fig. 3. Result of the Register Analyzer Framework for the BS Case Study

TABLE 1. SUMMARY OF SELECTIVE SWIFT-R IN EACH CASE STUDIES

Case Study	Most Vulnerable register
BC	R2
BS	R3
MM	R5
QS	R0

After selecting the register, we will protect the register by choosing the particular Selective Swift-R method. In Fig 4, the result of injecting faults is shown in the dark blue for the unprotected results, and in the light-orange for the protected ones. The improvement in each case study is for BC 16.97%, for BS 10.65%, for QS 9.30% and for MM 10.10%. The critical point to mention is that we are only protecting one register in each case study. This method is well suited for when we have limitation and can't use a large number of registers and memory. Because as is mentioned before, the software methods to protect the registers suffer from high overhead in code size. However, now that the most vulnerable register is determined with the help of our framework, it alone can be protected, reducing the code size dramatically. As depicted in Fig. 5, The code size of the protected case studies has increased with a factor of 1.3 for BC, 1.2 for BS, 1.54 for QS and 1.98 for MM compared to the unprotected versions.

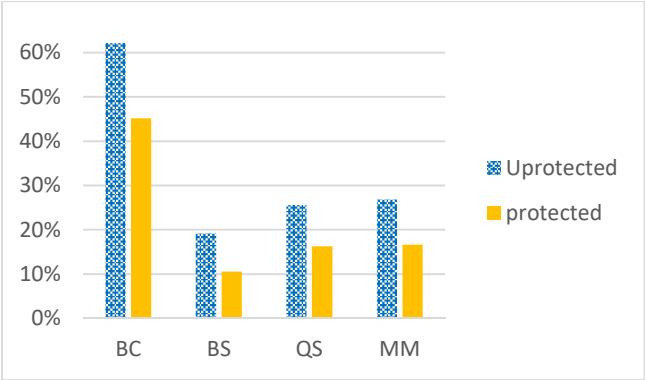

Fig. 4. The percentage of injected error resulting SDC value in four Case Studies

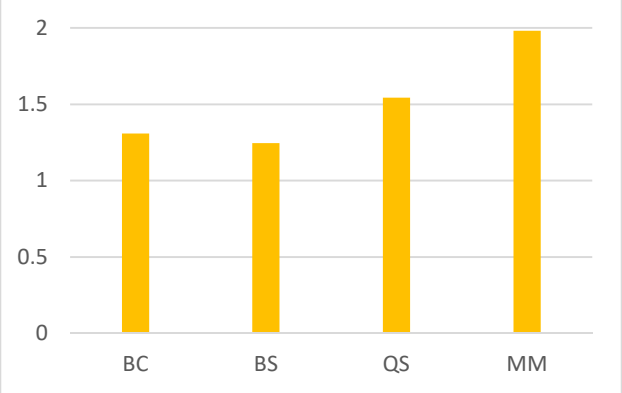

Fig. 5. Code Size Overhead in protected Case Studies relative to the unprotected ones

V. CONCLUSION

This research offers a methodology for monitoring and analyzing the interactions at run-time for program registers' to memory. As a result, this framework generates an ordered list indicating the vulnerability per register. Based on this list, the most vulnerable register is protected by implementing Selective SWIFT-R. This selective software-based fault tolerance strategy provides protection against soft errors while imposing a minimal overhead. This makes it well suited for COTS processors. To determine the impact of Selective SWIFT-R, fault injection experiments have been conducted and the imposed code size overhead has been measured. To further expand this study, a compiler extension will be developed that can automatically implement Selective SWIFT-R by using the results of our Register Analyzer framework. This will enable considering a higher amount and more diverse case studies.

REFERENCES

[1] R Edwards, C Dyer, and E Normand. Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics. In IEEE Radiation Effects Data Workshop (REDW), pages 1-5. IEEE, 2004.

[2] M. Nicolaidis, Ed., Soft Error in Modern Electronic System, ser. Frontiers in Electronic Testing. Springer US, 2011, vol. 41.[Online]Avaible:http://www.springer.com/us/book/9781441969927

[3] Leonardo M. Reyneri, Claudio Sanso_e, Claudio Passerone, Stefano Speretta, Maurizio Tranchero, Marco Borri, and Dante Del Corso. Aerospace Technologies Advancements, chapter 9: Design Solutions for Modular Satellite Architectures. Intech, Olajnica 19/2, 32000 Vukovar, Croatia, January 2010.

[4] M. Pignol. COTS-based Applications in Space Avionics. In Proc. 13th Design, Automation and Test in Europe conf., DATE, pages 1213-1219. Dresden, Germany March 2010.

[5] Michael Nicolaidis. Soft Errors in Modern Electronic Systems, volume 41 of Frontiers in Electronic Testing. Springer, 1st edition, 2011.

[6] M. Pignol. How to cope with SEU/SET at system level? In Proc. 11th IEEE Int. On-Line Testing Symp, IOLTS, pages 315 { 318, July 2005.

[7] S. Goloubeva, Rebaudengo and Violante, Eds., Software-Implemented Hardware Fault Tolerance. Springer, 2006. [Online].Available:http://link.springer.com/book/10.1007%2F0-387-32937-4

[8] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A.Tambara, P. Rech, and H. Quinn, "S-seta: Selective software-only error detection technique using assertions," IEEE Transactions on Nuclear Science, vol. 62, no. 6, pp. 3088–3095, Dec 2015.

[9] J. Isaza-González, F. Restrepo-Calle, A. Martínez-Álvarez and S. Cuenca-Asensi, "SHARC: An efficient metric for selective protection of software against soft errors", *Microelectronics Reliability*, vol. 88-90, pp. 903-908, 2018. Available: 10.1016/j.microrel.2018.07.008.

[10] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A.Tambara, P. Rech, and H. Quinn, "S-seta: Selective software-only error detection technique using assertions," IEEE Transactions on Nuclear Science, vol. 62, no. 6, pp. 3088–3095, Dec 2015.

[11] M. Nicolaidis. Design for soft error mitigation. IEEE Trans. Device Mat. Reliable., 5(3):405 - 418, Sept 2005.

[12] G. A Reis, J. Chang, and D. I August. Automatic instruction-level software-only recovery. IEEE Micro, 27(1):36 - 47, jan-feb 2007.

[13] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Software-Implemented Hardware Fault Tolerance, volume XIV. Springer, 2006.

[14] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In Proc. Int. Conf. on Dependable Systems and Networks, DSN, pages 71-78, 2000.

[15] C. Bolchini. A software methodology for detecting hardware faults in VLIW data paths. IEEE Trans. Reliab.,52(4):458-468, dec. 2003.

[16] S. Raasch, A. Biswas, J. Stephan, P. Racunas, and J. Emer,"A fast and accurate analytical technique to compute the avf of sequential bits in a processor," in Proceedings of the 48th International Symposium on Microarchitecture, ser. MICRO-48. NewYork, NY, USA: ACM, 2015, pp. 738–749. [Online].Available:http://doi.acm.org/10.1145/2830772.2830829

[17] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. Proc. First IEEE Int. Work-shop on Source Code Analysis and Manipulation, pages 33 - 42, 2001.

[18] M. Rebaudengo, M. S. Reorda, and M Violante. A new approach to software-implemented fault tolerance. J. Electron. Test., 20(4):433-437, Aug 2004.

[19] O. Ruano, J.A. Maestro, and P. Reviriego. A Methodology for Automatic Insertion of Selective TMR in Digital Circuits A_ected by SEUs. IEEE Trans. Nucl. Sci.,56(4):2091-2102, August 2009.

[20] B. Pratt, M. Carey, J.F. Carroll, P. Graham, K. Morgan, and M. Wirthlin. Fine-Grain SEU Mitigation for FPGAs Using Partial TMR. IEEE Trans. Nucl. Sci.55(4):2274-2280, August 2008.

[21] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. ACM Trans. Graph., 29(1):5:1-5:11, December 2009.

[22] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A.Tambara, P. Rech, and H. Quinn, "S-seta: Selective software-only error detection technique using assertions," IEEE Transactions on Nuclear Science, vol. 62, no. 6, pp. 3088–3095, Dec 2015.

[23] A. Reis, J. Chang, and D. I August. Automatic instruction-level software-only recovery. IEEE Micro, 27(1):36-47, jan-feb 2007.

[24] Imperas. Revolutionizing embedded software development. Online.[Online]. 2018. Available: http://www.imperas.com/

[25] J. Vankeirsbilck, J. Van Waes, H. Hallez and J. Boydens, "Impact of selectively implementing control flow error detection techniques", *Internet of Things*, vol. 12, p. 100260, 2020. Available: 10.1016/j.iot.2020.100260.

[26] Guang, Z. Zhang, Linear Network Error Correction Coding, 1st Edition, SpringerBriefs in Computer Science, SpringerVerlag New York, 2014. doi:10.1007/978-1-4939-0588

[27] R. Zhu, Y. Ma, Information Engineering and Applications, 1st Edition, Vol. 154 of Lecture Notes in Electrical Engineering, Spinger-Verlag London, 2012

[28] R. C. Gonzalez, R. E. Woods, Digital Image Processing, Pearson, 2017.