Selective Duplication and Selective Comparison for Data Flow Error Detection

Venu Babu Thati, Jens Vankeirsbilck, Jeroen Boydens
Dept. of Computer Science, M-Group
KU Leuven Bruges Campus
8200 Brugge, Belgium
(Venubabu.Thati—Jens.Vankeirsbilck—Jeroen.Boydens)@kuleuven.be

Davy Pissort
Dept. of Electrical Engineering, M-Group
KU Leuven Bruges Campus
8200 Brugge, Belgium
Davy.Pissoort@kuleuven.be

***Abstract*—Embedded systems' hardware can be impacted by soft errors, which can cause data flow errors in the systems' software. In this paper, we present a novel software-based approach to counter data flow errors, called Selective Duplication and Selective Comparison (SDSC). First, we validated our SDSC technique by implementing it for six case studies and submitting it to a fault injection campaign. Next, we measured its execution time overhead. To put the measured results into perspective, we compared them to those of two established techniques, called Critical Block Duplication (CBD) and near Zero silent Data Corruption (nZDC). The results show that our SDSC technique has a higher error detection ratio with a lower silent data corruption compared to both the CBD and nZDC techniques. This does, however, come with a slightly higher execution time overhead.**

***Index Terms*—software reliability, fault tolerance, soft errors, fault detection**

I. Introduction

In recent years, embedded systems have been used more and more in, amongst others, Internet-of-Things, Industry 4.0, mechatronics, safety-critical and general purpose applications. This has been made possible due to technological advances in semiconductor industry such as, increasing computing capacity and lowering the required power supply. On the opposite, a disadvantage of these technological trends is that harsher working environments have been created for embedded systems, which makes them more susceptible to external disturbances [3], [17]. For example, the fact that embedded systems are close to one another together with increased usage of wireless communication could induce errors due to Electro-Magnetic Interference (EMI) [2], electrical noise [4], high energy particle radiation [7], temperature fluctuation [10], etc.. These disturbances can introduce erroneous bit-flips (i.e a logical 0 changing to logical 1 or vice-versa) in the system's hardware.

Furthermore, bit-flips can have impact on the executing software and can cause either Control Flow Errors (CFEs) or Data Flow Errors (DFEs). A CFE is the corruption of the execution order of instructions. A DFE is the corruption of input, intermediate or output data. Both types of errors can lead to improper handling of actuators, can cause the program to hang or even to crash in some scenarios. This makes it imperative to have a fault tolerant mechanism that counters CFEs and DFEs. This paper focuses on the latter.

There are two types of fault tolerance techniques that aim to protect processors from bit-flips: hardware-based and software-based techniques. Hardware-based fault tolerant techniques depend on duplicating or adding hardware modules. They usually change the original processor architecture by adding logic redundancy and majority voters [5]. Although they are quite performant, they also exhibit a significant increase in power consumption and extra design and manufacturing costs per system produced.

On the other hand, software-based fault tolerant techniques are popular and well known to protect systems against bit-flips by including extra code to the original program [6], [11]. Software-based fault tolerant techniques can be implemented with a limited increase in development time. Although software-based mechanisms add fault-tolerance to the system, they require extra execution time and extra memory due to the execution of the extra code. In the interest of their cost and flexibility, these software solutions are used in numerous applications [9], [14], [16].

As mentioned before, this paper focuses on DFEs. To counter DFEs, numerous software-based techniques have been proposed and implemented for several years [6], [12], [18]. Duplication and comparison is the most used mechanism to detect data flow errors. Based on the mechanisms involved in each existing technique, these techniques are further divided into two categories: full duplication and selective duplication. In full duplication techniques, the entire code base is duplicated and their results are compared to report the errors. With full duplication, maximum fault detection ratio with more imposed overhead can be achieved. On the other hand, in selective duplication techniques, only that part of a code identified as the most vulnerable is duplicated and their results are compared to report the errors. Decrease in overhead with a reduced fault detection ratio can be achieved with selective duplication [18]. To address the drawbacks from full and selective duplication mechanisms, this paper proposes a novel data flow error detection technique Selective Duplication and Selective Comparison (SDSC).

The remainder of the paper is organized as follows. Section II describes some related work. Section III discusses the newly proposed SDSC technique. Section IV discusses the experimental setup. Section V presents the results obtained through experiments. Section VI describes our future work plans and conclusions are drawn in Section VII.

II. Related Work

This paper focuses on selective duplication mechanisms. Most existing selective duplication techniques are based on control flow graph (CFG) analysis to identify vulnerabilities in the program. The CFG is a representation of a program and its flow, in which the program is divided into a number of basic blocks and edges. A basic block is a sequence of consecutive instructions with exactly one entry and one exit point [20]. To evaluate the capabilities of our newly proposed selective duplication mechanism, we performed fault injection experiments and compared it with some established and recently proposed techniques: Critical Block Duplication (CBD) [1] and near Zero silent Data Corruption (nZDC) [6].

CBD proved to be the best technique for achieving a better fault detection ratio among other existing selective code duplication techniques [18], making it worthy of comparison with our methodology. nZDC is another recently proposed technique which produces less silent data corruption (SDC) according to the Didehban et al. [6]. Furthermore, nZDC is a hybrid technique to detect both CFEs and DFEs. For appropriate comparison with our DFE based SDSC, in this work, we only considered DFE based mechanism involved in nZDC.

*1) **Critical Block Duplication**:* To implement the CBD technique, critical basic blocks have to be identified in the CFG [1]. Abdi et al. defines critical basic blocks as the basic blocks with the highest number of fan-outs. This consideration is based on their results being propagated to many parts of program and its affect on other blocks [1]. Once those critical basic blocks are identified, the instructions of critical basic blocks are duplicated and comparison instructions are inserted to compare the results of the original operations with their duplicated instructions. Any mismatch between the results indicates a DFE occurrence.

*2) **near Zero silent Data Corruption**:* nZDC introduces the concept of checking load instructions to make sure store instructions are executed fault free [6]. The main idea involved in this technique is to load back the stored value from memory and check that against the stored value [6]. Memory read instructions are the most frequent unprotected instructions in some existing techniques for data flow error detection [13], [21]. If memory read instructions are corrupt, the program execution can go wrong and compare instructions are unable to detect any errors. In such a case, to protect those instructions, Didehban et al. used load instruction duplication, both in memory read instructions as well as logical and computational instructions. An error is reported upon a mismatch between the result comparisons.

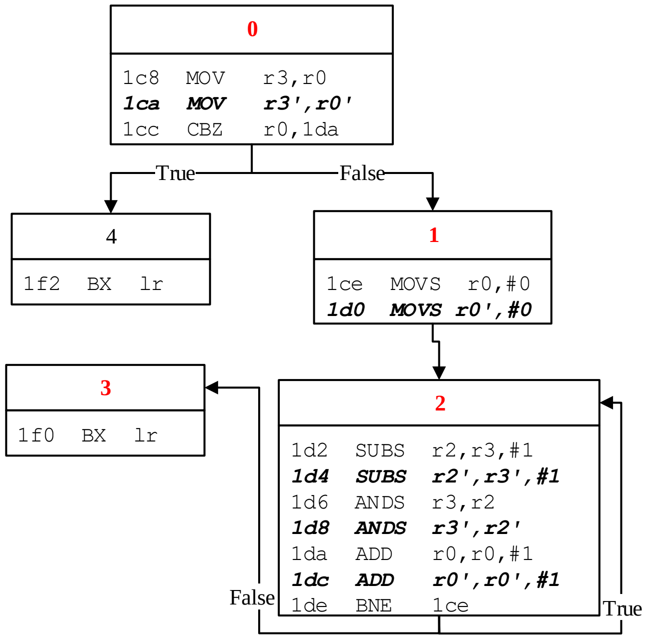
III. Selective Duplication and Selective Comparison

Our SDSC technique relies on selective duplication and selective comparison mechanisms. As the name gives away, we opt to duplicate a selective part of original program and place the necessary comparison instructions in a few selected basic blocks. Fig. 1 shows an example CFG, containing 5 basic blocks, in the ARMv7-M assembly language.

A. Selective Duplication

In our selective duplication mechanism, we propose a new strategy called Vulnerable Path Duplication (VPD). We define the vulnerable path as the longest path in a CFG because of the largest possibility of error occurrence in such a path. Furthermore, blocks which are presented in that longest path are more likely to be vulnerable ones. By protecting those selective blocks, reduction in overhead, but still keeping high fault detection ration, can be achieved.

An example for longest CFG path identification and duplication from CFG is shown in Fig. 1. Analyzing Fig. 1, program execution through blocks 0, 1, 2 and 3 (highlighted in red) form the longest path. These blocks will be duplicated to detect data flow errors. In certain cases there could be multiple longest paths in a CFG, where there are two or more paths with exactly the same number of basic blocks in them. In such a case, the longest path is chosen based on the higher number of memory access instructions in the path. Suppose, in a few rare case scenarios, that the number of memory access instructions are also the same, then selection happens based on the highest number of data processing instructions present in each of the paths. In case that the number of data processing instructions would also match, the choice of the longest path will depend on the higher number of cumulative instructions. Choice of path upon further equalities is subjective to the path that has been identified first.



```
0
1c8  MOV   r3,r0
1ca  MOV   r3',r0'
1cc  CBZ   r0,1da

4
1f2  BX  lr

1
1ce  MOVS  r0,#0
1d0  MOVS  r0',#0

3
1f0  BX  lr

2
1d2  SUBS  r2,r3,#1
1d4  SUBS  r2',r3',#1
1d6  ANDS  r3,r2
1d8  ANDS  r3',r2'
1da  ADD   r0,r0,#1
1dc  ADD   r0',r0',#1
1de  BNE   1ce
```

Fig. 1. Illustration of selective duplication mechanism with sample CFG.

Duplication of branch instruction leads to wrong execution order or program crash. Therefore, according to Fig. 1, although block 3 is located in the longest path it is not duplicated. The duplicated instructions are represented in bold *italic* form as can be seen in Fig. 1.

Further, in duplicated instructions, the duplicated registers are indicated with inverted comma ('), meaning that r0', r2', and r3' are the duplicate of registers r0, r2, and r3 from the original instructions. Moreover, as given in Fig. 1, and later also in Fig. 2, ADD, SUB, MOV, CBZ and CMP, indicate assembly-level instructions for addition, subtraction, move, compare and branch on zero, and compare operations, respectively. Likewise, BX represents branch indirect, whereas BNE represents a branch instruction which is not equal to a condition.

B. Selective Comparison

Comparison is equally important as duplication to detect data flow errors. A wide variety of possibilities to insert comparison instructions exist. However, it is not easy to choose a position to insert those instructions, because sometimes they could lead to significant extra overhead. In SDSC, we only compare the results of the original and duplicated instructions present in the critical basic blocks. We define critical basic blocks as those blocks with two or more incoming edges in the previously defined vulnerable path of the CFG. These critical basic blocks are more likely to be included in many other execution paths throughout the target algorithm. If those blocks are well protected, probability of the error propagation to the other blocks can be reduced.

Fig. 2 depicts an example of how the critical blocks are identified with the sample CFG. Analyzing Fig. 2, basic block 2 is identified as a critical block as it has two incoming edges and is also a part of the longest path in the CFG. Inserting comparison instructions only in the critical blocks, here the basic block 2, ensures a reduction in execution time overhead. The comparison instructions are represented in **bold**.

IV. Experimental Setup

We applied our SDSC protection mechanism to the six different case studies and executed them on an ARM Cortex-M3 driven microcontroller. This section describes the case studies and fault injection process used to perform experiments.

A. Case Studies

We selected six different case studies to implement and test both our proposed SDSC technique and the CBD and nZDC techniques. These case studies are implementations of: bubble sort (BS), cyclic redundancy check (CRC), selection sort (SS), matrix multiplication (MM), Dijkstra (DIJ) and insertion sort (IS) algorithms. Being highly used as validation case studies in existing literature for data flow error detection techniques, these case studies also have varying complexity and CFGs as shown by Vankeirsbilck et al. [20]. This makes them highly suitable for experiments to assure a thorough evaluation of our SDSC technique. Table I indicates amounts

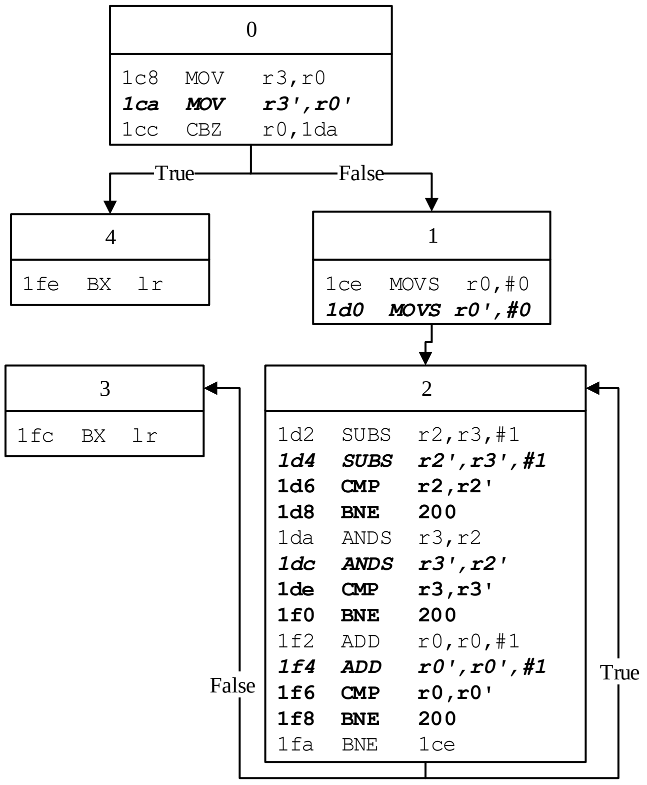


Fig. 2. Illustration of SDSC mechanism with sample CFG.

of original, duplicated and comparison instructions used for the different case studies.

Different sorting algorithms were chosen for test and validation since these are often used in a variety of applications. For example, to perform a big data analysis and interpretation, one needs to rely on sorting of information to perform fast analysis [9], [19], [22]. MM algorithm was considered because it is used in various embedded domains such as image processing and compression, robotics [8], [19]. The extensive application based overview of CRC and its usage is well explained by Guthaus et al. [9]. Further, DIJ algorithm calculates the shortest path between different nodes, which is highly used in routing applications [15], [20]. Among the six considered case studies, CRC and DIJ are from the MiBench [9].

B. Fault Injection Process

To validate the implementations and to enable a thorough DFE injection, we used our in-house developed fault injection process. This process progressively steps through the target algorithm and injects all possible faults for that location. The used fault model for the evaluation is a single-bit bit-flip model which is the most used fault model in this type of research and supporting literature. Moreover, it is known that most systems already have some countermeasures in place for their memory like parity and ECC but not for processor registers. For this reason we only inject bit-flips in processor registers.

In the DFE injection process, for each encountered instruction, we analyzed the instructions for registers they use. Then,

TABLE I
NUMBER OF ORIGINAL, DUPLICATED AND COMPARISON INSTRUCTIONS FOR EACH SELECTED CASE STUDY OF SDSC, CBD AND nZDC TECHNIQUES

Case study	**Original instructions**	**SDSC**		**CBD**		**nZDC**	
		Duplicated instructions	***Compare instructions***	***Duplicated instructions***	***Compare instructions***	***Duplicated instructions***	***Compare instructions***
BS	25	12	20	8	18	9	14
CRC	17	10	18	7	16	9	12
SS	50	35	54	27	68	34	48
MM	63	47	62	22	52	46	68
DIJ	87	62	110	41	92	58	80
IS	46	21	22	21	44	20	26

for each register, we independently inject a bit-flip for each possible position. E.g. in a 32-bit architecture, we inject 32 bit-flips per encountered register per instruction. Overall, 20000 DFEs were injected per each implemented case study of our experiments.

Each of the injected DFEs can have one of the following four effects:

Software Detected Fault (SDF): The injected fault is detected by the duplication and comparison instructions used in the implemented mechanism for data error detection.

Hardware Detected Fault (HDF): The injected fault is detected by the default hardware fault exception handler, in this case the ARM Cortex processor family.

No Effect Fault (NEF): The injected fault is not detected and it does not have an effect on the output of the program.

Silent Data Corruption (SDC): The injected fault is not detected by the implemented mechanism and it changes the output of the program. As this is the most disastrous fault we focus on the reduction of these type of faults in our SDSC technique.

V. RESULTS AND DISCUSSION

This section reports and discusses the results of our experiments which were conducted to evaluate the proposed technique. To evaluate our SDSC, we compare it with existing CBD and nZDC techniques. First, we compare the three techniques based on the results of the fault injection campaign. Next, we compare them based on their imposed execution time overhead.

A. *Fault Injection Results*

Fig. 3 displays the results obtained from the fault injection campaign. The obtained results are in the form of four different fault categories: SDF, HDF, NEF and SDC. Analyzing Fig. 3, the most interesting fault category for comparison is the one of the faults detected by our SDSC technique in the form of SDF. It is clear from the first fault category that SDSC displays an increase in SDF for 5 out of 6 considered case studies because of its accurate selective duplication and comparison mechanism. In comparison with CBD and nZDC, the average percentage of SDF in SDSC has increased by 13.6% and 50.5%, respectively. In Fig. 3, the nZDC technique shows a low SDF ratio for the CRC technique. This is because the comparison mechanism in nZDC is located only after store and load instructions. In essence, CRC does not have store instructions to place comparisons as a result very few number of comparisons were inserted.

In addition to the SDF, the percentage of faults detected in the HDF category is much lower among all case studies using all techniques. In this category, faults are not detected by neither our SDSC or CBD and nZDC, but detected by the default hardware fault exception handlers. That is why this category is less important in our comparison.

The third fault category is referred to as NEF. This category indicates the faults which were undetected and did not have an effect on the output of the program. Therefore, this category can be omitted out of the comparison.

The final and most critical fault category is referred to as SDC. It represents those faults that were not detected by the implemented data error detection technique SDSC or CBD and nZDC, and that were able to corrupt the output of the case study. In this case, our SDSC shows a lower number of undetected faults for most case studies in comparison with CBD and nZDC. The average SDC with our SDSC is

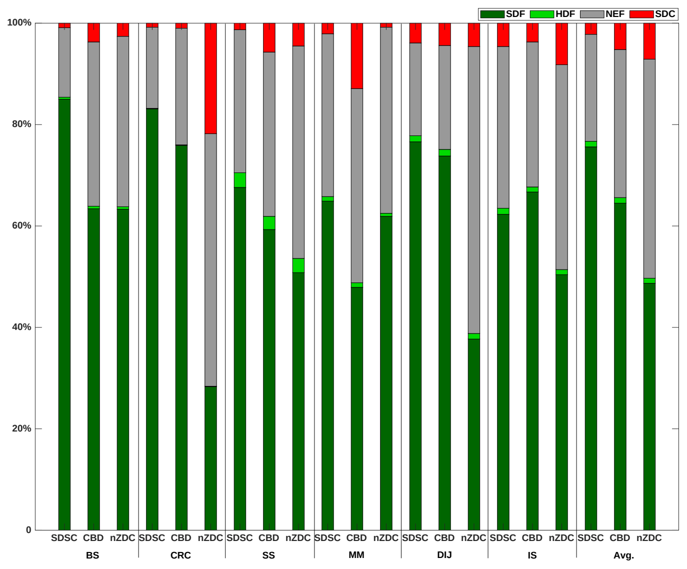


Fig. 3. Fault injection results of SDSC, CBD and nZDC techniques.

only 2.3% whereas 5.3% and 7.1% with CBD and nZDC, respectively. The decrease of SDC in our SDSC is mainly because of the accurate selective protection of the code.

B. Execution Time Overhead

Erroneous bit-flip detection does not come cheap. Counter measures do introduce execution time overhead. Execution time overhead expresses the extra time it takes for the algorithm to execute in an error free run. In this study, the introduced execution time overhead (ETO) is calculated based on the following two parameters; protected execution time (PET) and unprotected execution time (UPET) as given in (1).

$$ETO = \frac{\text{PET} - \text{UPET}}{\text{UPET}} \quad (1)$$

Fig. 4 shows the imposed execution time overhead of SDSC, CBD and nZDC. Analyzing Fig. 4, it is clear that our SDSC shows a slight increase in overhead for most case studies in comparison with CBD and nZDC. In more detail, comparison with CBD and nZDC, results show that average execution time overhead of our SDSC is increased by 20.3%. This percentage is quite small in comparison with the improvements achieved through the two critical factors such as SDF and SDC. Moreover, as microcontrollers are having evermore computing capacity, impact of this execution time overhead will reduce.

VI. Future Work

In this paper, we have discussed and presented a technique for data flow error detection. As aforementioned, there exist numerous techniques for control flow error detection separately. However, only very few techniques exist on the combination of data flow and control error detection. We would like to merge our SDSC technique with one of the recently proposed better techniques for control flow error detection. In addition, we would also like to validate our new experiments with more case studies to confirm the effectiveness of this new technique.

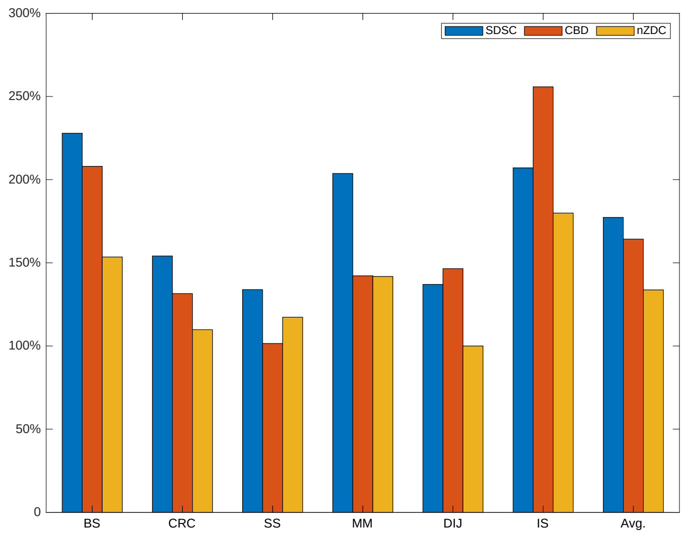


Fig. 4. Execution time overhead of SDSC, CBD and nZDC techniques.

VII. Conclusion

In this paper, we presented a new technique for data flow error detection based on selective duplication and selective comparison mechanisms. SDSC was implemented on six different case studies. Then fault injection experiments were performed for those case studies. The performed experiments measured two criteria, namely fault detection capabilities and execution time overhead. The two measured criteria were then used to evaluate our SDSC technique by comparing it with two previous existing techniques namely CBD and nZDC.

The experimental results showed that SDSC detected the most data flow errors for all considered case studies consistently. In comparison with CBD and nZDC, the percentage of software detected faults in our SDSC has improved by 13.6% and 50.5%, respectively. In the same way, the undetected data flow errors with our SDSC is reduced to only 2.3% but with CBD and nZDC, it is quite high as 5.3% and 7.1%, respectively. On the other hand, in our SDSC, the execution time overhead has increased slightly which is quite small in comparison with the improvements achieved through SDF and SDC. As microcontrollers are executing faster and faster, we expect that this overhead can be neglected in the near future.

References

[1] A. Abdi, S. Asghari, S. Pourmozaffari, H. Taheri, and H. Pedram, "An optimum instruction level method for soft error detection," *International Review on Computers and Software*, vol. 7, no. 2, pp. 637–641, 2012.

[2] K. Armstrong, D. Pissoort, A. Degraeve, and J. Lannoo, "Risk management of electromagnetic disturbances," in *2018 IEEE International Symposium on Electromagnetic Compatibility and 2018 IEEE Asia-Pacific Symposium on Electromagnetic Compatibility (EMC/APEMC)*. IEEE, 2018, pp. 193–198.

[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[5] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar *et al.*, "Reliability on ARM processors against soft errors through SIHFT techniques," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2208–2216, 2016.

[6] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near zero silent data corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[7] M. J. Gadlage, A. H. Roach, A. R. Duncan, A. M. Williams, D. P. Bossev, and M. J. Kay, "Soft errors induced by high-energy electrons," *IEEE Transactions on Device and Materials Reliability*, vol. 17, no. 1, pp. 157–162, 2017.

[8] R. C. Gonzalez and R. E. Woods, "Digital image processing," 2012.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization, WWC-4*. IEEE, 2001, pp. 3–14.

[10] S. Jagannathan, Z. Diggins, N. Mahatme, T. Loveless, B. Bhuva, S. Wen, R. Wong, and L. Massengill, "Temperature dependence of soft error rate in flip-flop designs," in *IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2012, pp. SE2.1–SE2.6.

[11] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, 2002.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software implemented fault tolerance,” in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[14] L. M. Reyneri, C. Sansoè, C. Passerone, S. Speretta, M. Tranchero, M. Borri, and D. Del Corso, “Design solutions for modular satellite architectures,” in *Aerospace Technologies Advancements*. IntechOpen, 2010.

[15] S. C. Satapathy, V. Bhateja, and A. Joshi, *Proceedings of the International Conference on Data Engineering and Communication Technology: ICDECT 2016*. Springer, 2016, vol. 2.

[16] P. P. Shirvani, N. Oh, E. J. Mccluskey, D. Wood, M. N. Lovellette, and K. Wood, “Software-implemented hardware fault tolerance experiments: COTS in space,” in *International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (NY)*. Citeseer, 2000.

[17] B. D. Sierawski, R. A. Reed, M. H. Mendenhall, R. A. Weller, R. D. Schrimpf, S.-J. Wen, R. Wong, N. Tam, and R. C. Baumann, “Effects of scaling on muon-induced soft errors,” in *2011 International Reliability Physics Symposium*. IEEE, 2011, pp. 3C.3.1–3C.3.6.

[18] V. B. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort, and J. Boydens, “CDFEDT: Comparison of data flow error detection techniques in embedded systems: an empirical study,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2018, pp. 23:1–23:9.

[19] ——, “An improved data error detection technique for dependable embedded software,” in *IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2018, pp. 213–220.

[20] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, “Random additive signature monitoring for control flow error detection,” *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, 2017.

[21] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, “DAFT: decoupled acyclic fault tolerance,” *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140, 2012.

[22] R. Zhu and Y. Ma, “Information engineering and applications,” in *International Conference on Information Engineering and Applications (IEA)*, vol. 5, no. 7, 2011.